On Understanding and Forecasting Fuzzers Performance with Static Analysis

Dongjia Zhang
EURECOM
Biot, France, zhangdo@eurecom.fr

Andrea Fioraldi
EURECOM
Biot, France, fioraldi@eurecom.fr

Davide Balzarotti
EURECOM
Biot, France, balzarot@eurecom.fr

Abstract

Fuzz testing, a technique for detecting critical software vulnerabilities, combines various methodologies from previous research to improve its effectiveness. For fuzzing practitioners, it is imperative to comprehend the effects of distinct techniques and select the ideal configuration customized to the program they need to test.

However, evaluating the individual contributions of these techniques is often very difficult. Prior research compared assembled fuzzers and studied their affinity with different programs. Nevertheless, assembled fuzzers cannot be easily broken down into independent components, and therefore, the evaluation does not clarify which technique explains the performance of the fuzzer. Without understanding the potential impact of integrating different fuzzing techniques, it becomes even more challenging to adjust the fuzzer configuration for different programs under test.

Our research tackles this challenge by introducing a novel approach that correlates static analysis features extracted at compile time with the performance results of various fuzzing techniques. Our method uses diverse metrics to uncover the relationship between the static attributes of a program and the dynamic runtime performance of fuzzers. The correlation analysis performed on 23 target applications reveals interesting relationships, such as power schedulers performing better with larger programs and context-sensitive feedback struggling with a large number of inputs.

This approach not only enhances our analytical understanding of fuzzing techniques, but also enables predictive capabilities. We show how a simple machine learning model can propose a fuzzer configuration customized for a particular program using information collected through static analysis. In 11 of our benchmark programs, fuzzers using the suggested configuration achieved the best improvement over the baseline compared to AFLplusplus, LibFuzzer and Honggfuzz.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Fuzzing, Static-Analysis

ACM Reference Format:
Dongjia Zhang, Andrea Fioraldi, and Davide Balzarotti. 2024. On Understanding and Forecasting Fuzzers Performance with Static Analysis. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670348

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670348

1 Introduction

Modern fuzzers are a testament to the evolution of software testing, embodying a myriad of complex techniques [21] combined to form sophisticated tools for vulnerability detection.

However, their current complexity and very rapid evolution pose problems to the users. Researchers routinely propose dozens of new techniques every year, often claiming that their solutions perform better than previous approaches in the majority of the targets they tested. In such an intricate landscape, evaluating individual contributions is becoming more and more difficult, and choosing the best fuzzer configuration for a given target is a formidable challenge.

Prior work [15, 22, 23, 35] performed end-to-end comparisons among existing tools to help with the selection of the optimal fuzzer for a user's specific needs. However, we claim that a deeper analytical understanding of each individual fuzzing technique can yield a more tailored, effective solution.

The challenge. Feedback-driven mutational fuzzers have become the golden standard for fuzz testing. AFL++, LibFuzzer, and HonggFuzz are three such fuzzers adopted by ossfuzz[40], the largest fuzzing platform operated by Google, which to date has identified over $10K$ vulnerabilities and $36K$ bugs in open source projects. The modern advancement of feedback-driven mutational fuzzing techniques [32] and algorithms include the **Scheduler**, the module to choose next testcase to fuzz, the **Feedback**, the module in charge of driving the fuzzer towards inputs that explore new program space, and the **Mutator**, the module that defines how the fuzzer modifies the input.

Modern fuzzers benefit from (a combination of) these techniques but it is challenging to break down a fuzzer performance into the individual contribution of the different parts, and to conclude, for instance, that "*Fuzzer A is better than Fuzzer B on a given target because of component X*".

LibAFL, recently presented by Fioraldi et al. [20], is offering the possibility of building fuzzers by combining and composing a number of orthogonal components. This framework offers the opportunity to perform more in-depth studies in which individual scheduler, feedback and mutator techniques can be compared on a common baseline.

Our Approach. In this paper, we propose a novel approach to correlate static analysis features extracted at compile time with the performance outcomes of diverse fuzzing techniques compared against the same baseline fuzzer. This approach stems from the idea that a better understanding of static attributes can shed light on

the dynamics of fuzzer performance and can help security analysts to make an informed decision on which fuzzer to pick.

Our methodology is based on a collection of 59 static features. By collecting characteristics such as data flow graph [3] and control flow graph [7] patterns, instruction types from LLVM IR [5], and the initial corpus metrics, we create a diverse set of features that reflect the program behavior. With this methodology, we can explore the intricate connection between static program attributes and the dynamic performance of fuzzers during runtime.

Our analysis is divided into two parts. In the first part, we selected 13 generic techniques from the three fuzzing techniques categories and implemented single-technique fuzzers that adopted each of the techniques. We then studied the correlations between the performance of these fuzzers and the program features. Our results show that most of the techniques we tested had a non-negligible correlation on several static features. Specifically, properties such as corpus features, initial coverage, API types used in the program, and the shape of the control flow graph affected fuzzer performances in different ways.

By leveraging these correlations, we move to the second part of our analysis, in which we try to solve the challenge of optimal fuzzer component selection. For this purpose, we trained a ML model to predict which fuzzing techniques will exhibit the most promising performance for a given target program. We then automatically assembled a fuzzer that uses such techniques and tested it on a new set of target programs, different from the ones we used for training. Our experiments show that our prediction fuzzer had the best improvement over the baseline among other fuzzers, including when compared with popular off-the-shelf fuzzer like AFL++, HonggFuzz, and LibFuzzer. In conclusion, the predictive capabilities of our model make it far easier for practitioners to select the fuzzer configuration that is more likely to succeed based on the static analysis of the target.

For the reproduction of our work, we published source code of our framework, and the experiment results at https://github.com/fuzzing-static-analysis/fuzzing-static-analysis.

2 Background

In this section, we briefly introduce how statistical analysis is employed in benchmarking fuzzers' performance and the use of LibAFL to build a common baseline for our experiments.

2.1 Statistical Analysis & Fuzzing

Several studies have employed statistical tests to advance our understanding of which factors influence the performance of a fuzzer and which methodology to use for their assessment. The seminal study on fuzzing evaluation conducted by Klees et al. [27] in 2018 highlighted a number of critical issues in the experimental evaluations of fuzzing techniques, strategies, and algorithms. The authors scrutinized 32 papers and identified problems in every experimental evaluation they considered. They then emphasized the necessity of a statistically robust experimental setup, advocating for a performance metric that considers the inherent randomness of fuzzing. The paper provided guidelines to enhance the reliability of experimental evaluations and called for the use of statistical tests to ensure that observed improvements in fuzzer performance are not merely due to chance. This work served as a clarion call for more scientifically rigorous evaluations in fuzz testing research.

In 2022, Böhme et al. [14] scrutinized the reliability of code coverage as a stand-in for fuzzer effectiveness by utilizing statistical tools to explore the relationship between coverage and bug-finding capabilities. The authors employed Spearman's rank correlation to assess the strength and direction of the relationship between these two variables, opting for this non-parametric measure due to the non-linear relationship they observed. Furthermore, they introduced the concept of inter-rater agreement using Cohen's κ [26] to evaluate the consistency between coverage-based and bug-finding-based measures of fuzzer performance. Their analysis, which demonstrated only moderate agreement, provides critical insights into the limitations of using coverage as the sole metric for fuzzer effectiveness, thereby informing future evaluations with a more accurate statistical foundation.

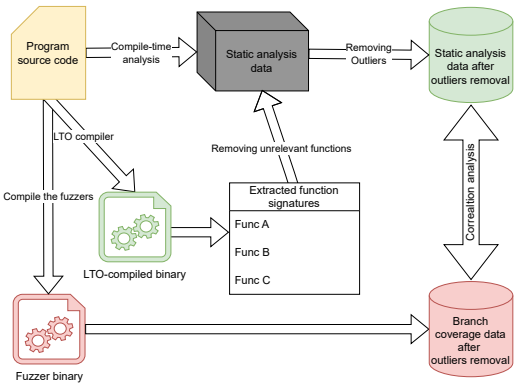
In the same year, Wolff et al. [47] pioneered non-parametric regression analysis to disentangle the effects of benchmark characteristics from the intrinsic performance of fuzzers. This approach, known as "explainable fuzzer evaluation" quantifies the degree to which a fuzzer's ranking is a product of the fuzzer's capabilities versus the properties of the benchmark used. Their statistical analysis revealed some dependencies, such as AFL's improved performance with benchmarks containing larger programs and LibFuzzer's decreased performance under similar conditions. This methodological framework also included **MLR (multiple linear regression)** to examine the combined influence of benchmark properties and fuzzer choice on the outcomes, providing a more nuanced view of fuzzer performance across different scenarios.

Our work continues along this line by addressing two main limitations of Wolff's approach. One results from the use of Spearman's rank correlation, which we show to provide unstable results. The other comes from the set of fuzzers used by Wolff et al.; the use of pre-assembled fuzzers, such as AFL++ and LibFuzzer, made it impossible to tell which part of the fuzzer contributed to the result. Our research tackles both issues and presents a more precise and comprehensive analysis of fuzzer correlations.

2.2 The LibAFL Framework

Wolff's research modeled how one fuzzer performed compared to another fuzzer within a given set. Our aim, instead, is to break down fuzzing into its individual components and investigate which part impacts its performance and correlates to some of the characteristics of the target application. Thus, we developed all our fuzzers from scratch by using LibAFL [20], which provides a modular framework and a set of customizable building blocks to create a custom fuzzer. The use of LibAFL facilitates finer comparisons between different fuzzers, as we can build a common baseline fuzzer and then swap out individual components, implementing alternative algorithms to produce a slightly modified fuzzer. When comparing the resulting fuzzers, the variation in performance can be attributed to the modified element, allowing for measurable differences in the performance of the specific component or algorithm. Essentially, our work breaks down fuzzers into smaller, modular components, evaluates them in relation to many different features of the target, and provides a solution for the limitation of Wolff's work.

Figure 1: The workflow of our correlation analysis



3 Methodology

This section outlines the approach we follow to establish a correlation between fuzzer performance and program features.

We begin by introducing the two key components of our study: the fuzzers and the program features. We first introduce the various types of fuzzers employed in our comparison and detail the the program features and its extraction process. We subsequently present the method for quantifying the difference in fuzzer performance derived from our experiments and explain the methodology employed in computing correlations. Figure 1 illustrates the workflow for our analysis.

3.1 Fuzzers Selection

Our objective is to determine how the characteristics of the target program affect the performance of fuzzing techniques. To this end, we studied various fuzzing techniques proposed in past fuzzing research. Based on that, we implemented a *naive* fuzzer to use as a baseline, and 12 other fuzzers adding different techniques on top of the naive fuzzer. These techniques affect three main components of the fuzzer:

Scheduler. The scheduler is in charge of two tasks. The first is called "power schedule" [12] which assigns power to the testcases in the corpus. Testcases that are given more power will undergo more trials when they are picked from the corpus. In this regard, we implemented the *fast* and the *explore* scheduling policies from AFLFast [13], and the *cov-accounting* technique proposed by TortoiseFuzz [46]. The second task of the scheduler is to select the next corpus to mutate. For this, we implemented the *weighted* scheduler from AFL++ [19], and the *rand-scheduler* (which randomly picks from the corpus for scheduling).

Feedback. The feedback is an essential component of the fuzzer that defines how and under what circumstances the fuzzer should save the mutated testcase back into the corpus. We implemented fuzzers with the ngram feedbacks for *ngram8* and *ngram4*, and with context-sensitive coverage feedbacks (*naive-ctx* fuzzer) from afl-sensitive[45]. We also included a fuzzer that implements the *value-profile* technique proposed by LibFuzzer [29]. This technique rewards the fuzzer whenever a comparison instruction has some novelties in its operands.

Mutator. Mutators define how the fuzzer mutates the inputs. For mutators, we implemented *mopt* from MOpt [31], *CmpLog* from AFL++ and RedQueen [9, 19], and the *grimoire* mutator from Grimoire fuzzer [11].

We opted not to adopt techniques proposed by nautilus [8] and gramatron [42] as these fuzzers are specialized for structured inputs, but not all our benchmark programs are designed to accept such inputs. Additionally, it would be impractical to prepare the grammar files required to start fuzzing for each of our benchmarks.

The *naive* fuzzer employs a basic scheduler, feedback, and a mutator inspired by AFL. The naive scheduler is designed to prioritize a policy to prefer quick and small testcases, while the naive feedback utilizes the standard branch coverage feedback with hitcounts [48]. Finally, the naive mutator uses a mutation called *havoc* mutation from AFL [48], which stacks different types of mutations, including bitflip, byteflip, arithmetic operations, token replacement, and others, to modify the input.

The greatest advantage of our approach is that by comparing these 12 fuzzers to the baseline *naive* fuzzer, we are able to precisely evaluate the effectiveness of each technique **independently** – while keeping the rest of the fuzzer constant.

Finally, in addition to these 13 custom fuzzers, we also added three of the most commonly used fuzzers, including AFL++ [19], LibFuzzer [30], and HonggFuzz [44].

3.2 Program Features Extraction

A critical phase of our approach is computing a number of program characteristics that may impact the fuzzers' performance. This feature extraction phase is carried out by conducting a static analysis that extracts features during compile time, primarily through information extracted from the LLVM IR.

First, we included a set of generic features, such as the program binary size, and features based on the type of instructions and their arguments. We then included new features that we *expect* to have correlations with the techniques we implemented. For example, *CmpLog* is equipped with the capabilities to break through the comparisons in the code, and *value-profile* tracks the novelties in comparisons. Therefore, we expect this fuzzer to correlate positively with the number of comparisons.

Last, we included features that were shown to have correlations by Wolff's work [47], such as seed size, initial coverage, and program size. We further expanded this set to better represent the program under test. For example, only the *initial coverage* was studied in the previous study; however, in our work, we dissected this feature into four types: the initial branch, line, function, and region coverage. These detailed features enable us to characterize the program in a more refined way.

All features are extracted for each individual function and are grouped into the following categories:

- **Generic features**, such as the program binary size, the number of the basic blocks, and the average nested level of the source code.
- **Instruction type**. As we extracted features from LLVM IR, we counted the frequency of each type of LLVM instruction, including `alloca`, `binary-operator`, `branch`, `call`, `cmp`, `load`, and `store` instructions.

- **Types of the instruction operand**. For all the LLVM instructions we analyzed, we count the occurrence of the types used in the operands of the instructions.
- **CFG and DDG**. We construct the CFG(Control flow graph) and DDG(data dependence graph) [3] of every function. From the CFG, we then extract features such as the existence of cycles (loop-back edges), the length of the shortest path, and the average length of the shortest paths in the CFG. Conversely, for DDG, we count the number of edges and nodes. We expect fuzzers with more fine-grained feedback, such as *ngram* fuzzer, to have correlations with these features as they will be rewarded for traversing different paths.
- **APIs** We extract two features for the APIs used in the program. The API calls related to memory allocation functions (such as malloc/free) and the calls to security-sensitive functions (such as memcpy and strcmp). These features are likely to be correlated with the *cov-accounting* fuzzer as it is designed to put more priority on functions with these API calls.
- **Comparisons** We grouped the comparison instructions in LLVM IR by the type of arguments, such as the comparison of integer, float, vector, and counted their frequency per function. In this way, we could conduct a detailed analysis of which type of comparison is involved in fuzzers' performance. These are essential features since the comparisons often block the fuzzer from progressing. We expect *cmplog* and *value-profile* to correlate with these features.
- **initial set of seeds** We also included the features representing the initial set of seeds. Although these characteristics are independent of the target program itself, they significantly influence the fuzzer's performance [25]. Therefore, we additionally calculate attributes linked to the initial seeds, such as the region, line, function, and branch coverage acquired through running these initial inputs.

After all features are extracted from each function, we prune out unnecessary analysis results from unreachable functions. This step is required as each fuzzing campaign is limited to explore only a part of the entire program. Fuzzers utilize a driver code, named *harness*, tailored for each fuzzer target to enter the program code through a specific API function and further explore other program parts. This fact means that functions not reachable from the harness code do not influence the fuzzing performance.

We compiled the programs using LLVM's LTO [1], or Link Time Optimization, compiler to achieve this goal. This optimization makes it possible to remove the unreachable code when the compiler links the object files into the final program binary. Additionally, we made sure that the compiler did not inline the functions in the code through optimization because we do not want to lose function signatures by inlining optimization. We then compared the function signatures in the binary compiled by the LTO compiler with those obtained by static analysis. If the function signature obtained from the static analysis does not exist in the LTO-compiled binary, we remove that function from the analysis result. This way, we ignore the functions that do not affect the fuzzing and remove their influence on the analysis.

3.3 Quantifying Differences among Fuzzer

We evaluated all fuzzers described above by measuring the branch coverage achieved after running several trials on different benchmarks. However, we need a way to quantify the difference between our fuzzers based on the results of the branch coverage results. The key point to consider is that we must compare the performance of fuzzers **across** different programs. Therefore, we cannot utilize the average of raw branch coverage values as a way to compare them. For instance, for large programs, there might be a significant difference in fuzzer performance, and therefore the variance of the fuzzer branch coverage outcome will be large. On the contrary, a small program may result in just slight differences in branch coverage between the fuzzer results.

To support the need for normalization, we will use a simple example, whose hypothetical results are presented in Table 1. In our example, we compare two fuzzers A and B on two target programs, a small program X and a larger program Y. We also suppose that our static analysis phase computed a certain feature F, which is lower for program X and higher for program Y.

If one would simply look at the average coverage, he could erroneously conclude that there is a positive correlation between the performance of fuzzer B and feature F. In fact, since program X is small, it is unlikely that there is a significant difference in the coverage of different fuzzers.

Table 1: Example on why normalization is needed

	Fuzzer A	**Fuzzer B**	**Feature F**
Program X	500	501	low
Program Y	2000	2500	high

To mitigate this problem we compute the **effect size** [43], which is a metric that measures the difference of values between two groups in results. Cohen's d [16] is a metric to measure the effect size, defined as:

$$d = \frac{\mu_1 - \mu_2}{s_p}$$

where μ_1 is the **mean** of the first group, μ_2 is the **mean** of the second group, and s_p is the **pooled standard deviation** [6] of the two groups.

Intuitively, the value of Cohen's d indicates the degree of difference between the two groups. A difference of one in this value can be interpreted as a difference of s_p in their average between the two groups. By dividing the difference in means by the pooled standard deviation, we normalize the variance of each test program.

We can use this value to quantify the difference between the fuzzer that implements each fuzzing technique and the baseline naive fuzzer. The effect size helps us answer the question, "How much difference did this technique bring compared to the naive fuzzer?" by evaluating the effect size.

We will repeat this for each individual technique and finally consider also AFL++, LibFuzzer, and HonggFuzz as example of complex pre-packaged solutions built by composing different techniques.

3.4 Correlation Analysis

To capture the correlations between performance and program features Wolff. et al. [47] computed the **Spearman's correlations** [24] between the features and the *ranking* of the coverage values rather

than the raw values themselves. It is essential to note that this approach based on rank's correlation can only provide the relative performance of one fuzzer over another (and thus, the results depend on which other fuzzers are used in the experiments) but cannot be used to assess the absolute performance of a particular fuzzer on a program with given characteristics.

Moreover, the information provided by the ranking can be deceiving. For instance, performing third out of 10 fuzzers could be an excellent result when the difference between first and third place is only a single edge, a case we observed in several of our experiments. In this scenario, the top three fuzzers should be considered roughly equivalent, as picking one over the others would not provide a clear advantage for a testing campaign. However, if between the second and third place, there was a large gap in coverage, choosing the third fuzzer would be clearly a poor choice.

Small differences in the coverage can also make the ranking unstable as the impact of randomness can make a fuzzer fluctuate between ranking positions.

So, while we computed the Spearman correlation with the fuzzer ranking to compare our findings with Böhme's results, we also compute the more robust **pearson's correlations** [24] between the different program features and the *effect size*. This mitigates the aforementioned problems and provides a better understanding of which techniques are more suitable to test a given program. However, since Pearson's correlation is susceptible to outliers, we computed the interquartile range for each of the program features and branch coverage results and removed the data points that lie outside the interquartile range. Since our methodology requires calculating the effect size compared to the baseline, we do not evaluate Pearson's correlation for comparing AFL++, LibFuzzer, and HonggFuzz. These fuzzers are not incremental additions to our baseline fuzzer, and therefore, measuring the effect of added techniques on top of the baseline through Cohen's effect size is invalid.

Finally, in addition to Spearman's rank correlation, we computed another rank correlation called Kendall's τ [39] for a more diversified evaluation of the correlation using three metrics.

4 Prediction

While correlation can provide useful insights, our final goal is to investigate the ability to *automatically* configure a fuzzer by choosing all its main components from a set of available approaches based on the characteristics of the target application. This task requires to predict the optimal configuration that would best fit the target program and is more likely to provide better results during a fuzzing campaign. The overview of our prediction workflow is illustrated in Figure 2.

4.1 Fuzzer Component Prediction

As we discussed in 3.1, we separated the fuzzing techniques into three categories, respectively dedicated to the **scheduler**, the **feedback**, and the **mutator**. Since the techniques in these three categories are orthogonal, we can combine them to build an optimal fuzzer configuration. However, the combination of different features makes the task more complex. For example, a target program may have numerous token comparisons, which can be effectively addressed using the *cmplog* algorithm. Similarly, it could begin with a large initial corpus where the *fast* scheduler is more dominant.

Therefore, we decided to use a **random forest** ML classifier to predict which combination would provide better results. In particular, we built a random forest classifier for each technique and trained them by providing the program features and the performance of the corresponding fuzzer as input. We did not use **MLR (multiple linear regression)** models because they are easily affected by outliers. Since we collected a diverse set of benchmarks, we prefer to avoid prediction models that are susceptible to extreme values in the data points. Whereas, one of the great advantages that random forest classifier offers is its robustness against these outliers [17]. We discuss the design choice of the prediction model further in section 8.2. The model estimates Cohen's effect size as the output using the input data. For the technique selection, we chose the one with the highest positive effect size in each technique category. To reduce the number of parameters, given the fact that the training data is limited, we pre-filtered the list of features we used to train the model based on the result from the correlation analysis, namely, only features correlated with at least one of the techniques in the same group were used as the training data for that group.

Based on this method, we built and trained the model with the programs in the same benchmark sets we used for correlation analysis. We discuss alternative design choice of the prediction models we considered in section 8.2, and possible limitation of our approach in Section 8.4.

4.2 Fuzzer Composition

Our prediction models output the best-performing component of the fuzzer for different programs. Next, we validated our approach's effectiveness by experimenting with an additional set of benchmarks. For this, we built a fuzzer based on the result of the ML prediction, i.e., by including the scheduler, feedback, and mutator selected by the classifier. We then ran our prediction-best fuzzer in conjunction with all the 12 single-technique fuzzers that only implement one technique at a time. We cannot evaluate every combination of the fuzzers because conducting experiments for all 120 ($= 6 \times 5 \times 4$) combinations for each program is not feasible due to resource limitations. By running the 12 single-techniques fuzzer, we can experimentally determine which technique *actually* works best for the program under test. Therefore, we can assess the accuracy of our predictions by comparing the real best technique.

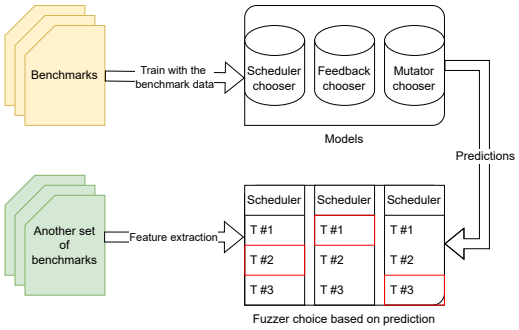
To gauge the performance of our prediction approach, we then evaluate our prediction fuzzer against the baseline fuzzer, along with AFL++, LibFuzzer, HonggFuzz.

5 Implementation

This section introduces the implementation we used for our approach. Our analysis framework comprises three parts: the data extraction pipeline, the comparison fuzzers, and the data analysis module.

Data Extraction. To extract data from the target programs, we compiled them while using a custom LLVM analysis pass. The pass dumped program features in JSON format for each function in the source code. To gather data dependency information from the

Figure 2: The workflow of our fuzzer component prediction



source code, we utilized the LLVM pass from DDFuzz [33]. Additionally, we also compiled the programs by using link time optimization, and then we extracted the function signatures from the resulting LTO binary with Angr [41]. All function signatures are then demangled with binutils's c++filt [2] and processed by a Python script designed to remove functions that were considered irrelevant.

For the features related to the initial set of seeds, we ran fuzzer with the initial seeds, and later, we collected the coverage with llvm-cov[4]. The data extraction pipeline is written in 2.2K lines of C++ and 1.3K lines of python3 code.

Fuzzers. We built our fuzzers by using LibAFL [20], a framework for building fuzzers written in Rust. The modular design of LibAFL allowed us to implement the baseline fuzzer, the 12 variations that adopt various fuzzing techniques, and the composition fuzzers suggested by our prediction algorithm. In total, our LibAFL fuzzers required 6.7K lines of Rust code.

Correlation Analysis and Prediction Model. We scripted 1.1K lines of Python code to perform the correlation analysis and build the prediction model. For the correlation analysis, we used the *stats* module from *scipy*, while the ML component was implemented by using the *RandomForestClassfier* [10] model from scikit-learn [36].

6 Evaluation

Our experiments are divided into two parts, the first dedicated to the correlation analysis and the second to the prediction. This section starts with the details of the experiment setup, followed by the presentation of the results and their interpretation.

6.1 Experimental Setup

We selected 23 benchmarks from fuzzbench [34] to conduct our correlation analysis. As explained in Section 3, we tested a total of 16 fuzzers: one baseline, 12 fuzzers implementing independent fuzzing techniques, and three popular off-the-shelf fuzzers. Each fuzzer was run on the 23 benchmarks for 23 hours each and a total of 20 trials with the initial seed setup the same as fuzzbench [34].

Since the 368 experiments (23×16) required 169,280 CPU hours, it was impossible for us to conduct a local experiment. Therefore, we asked Google's fuzzbench team to run the experiment on Google Cloud's Compute Engine. Nine experiments could not be completed, three due to failures of HonggFuzz, and six due to the error in the AFL++ LLVM pass instrumentation we used in the *ngram* and *naive-ctx* fuzzers.[1]. Therefore, we ignored these failing cases for the correlation analysis stage and performed our analysis based on the results of the remaining 359 experiments.

The same dataset was also used to train our classifier, which was then tested on 11 additional targets selected from the SBFT'23 competition benchmarks [28]. Also, in this case, each fuzzer was executed for 23 hours and 20 trials.

We further evaluated the overhead for the feature extraction phase compared to the normal build. The first stage of extracting the feature using LLVM Pass has 26% overhead compared to the normal build, and the second stage of building the LTO binary has 16% more overhead. In total, it took us 242% more time to compile. The overhead of our analysis instrumentation is not high; however, because we have to build the target program twice, the resulting build time is about 2.4 times the normal build.

6.2 Correlation Analysis: Effect size

We started by computing Pearson's correlation by using the 23 raw data points (one per target program) for every feature, and the Cohen's d value compared to the naive baseline fuzzer. It is important to note that while we had 20 different trials for each target when we calculate Cohen's d , all 20 trials are reduced to a single value. Therefore, we have 23 data points for each feature-to-fuzzer correlation.

As explained in 3.4, we then removed the outliers from these datapoints. This is an important pre-processing step before conducting the pearson's analysis. We removed 1.42 data points out of 23 on average across all the 59 features.

We selected the results that meet two conditions. First, the correlation coefficient R-value must be higher than 0.40 or lower than -0.40. Correlation coefficients larger than this threshold are considered to be at least *moderately strong* [38]. Second, we only retained results with p-values lower than 0.05.

Out of the 59 features we extracted from the target programs, 30 of them, listed in Table 2, satisfied the aforementioned criteria and were, therefore, at least moderately correlated to at least one of the 12 techniques we tested in our study. We listed the interpretation of all the extracted features in Appendix A. Out of these 30, three were related to the program CFG, one to the DDG, two to the invoked APIs, four to the comparison types, five to the operands, seven to the instructions, six to the initial seeds, and two to general features of the program (number of basic blocks and program size).

To answer the question of what correlations exist for different fuzzers and programs, we present the individual results grouped by category, and summarize the key takeaway at the end. Some of the graphs illustrating the correlation result are shown in Figure 3.

Along with Pearson's R, we also present Spearman's R and Kendall's τ [39] for a multifaceted unbiased evaluation. These are the alternative metrics to evaluate the rank correlations, and if the correlation result from all these metrics agrees, then the correlation is solid. All these correlation values range from -1 to 1; a negative number means negative and a positive number means positive correlations. The interpretation of the correlation is slightly different between Pearson's R to Spearman'R and Kendall's τ. We detailed how we compute the Spearman's rank correlation later in

[1]The bugs have been reported and acknowledged by the developers https://github.com/AFLplusplus/AFLplusplus/issues/1855

Figure 3: From the left, Correlation between `bbs` and *fast*, between `i64_cmps` and *value-profile*, between `rg_cov` and *cmplog*, and between `m_ap` and *cov-accounting*

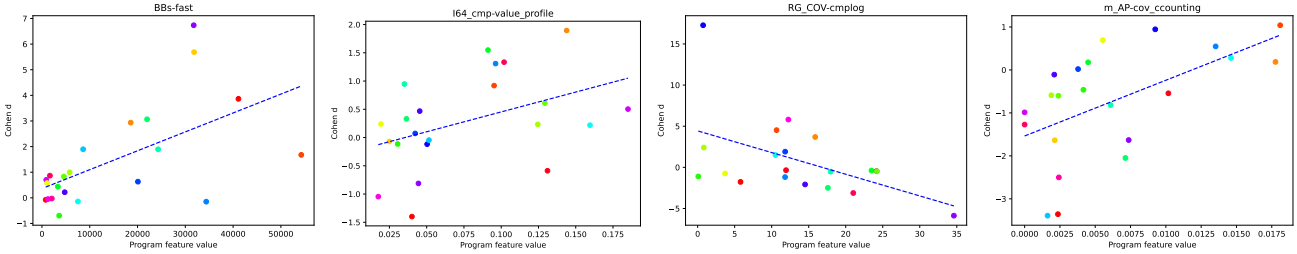


Table 2: Correlated Features

Feature Group	Correlated Features
Generic	`bbs`, `size`
Initial Seeds	`corpus`, `corpus_sz`, `br_cov`, `ln_cov`, `fn_cov`, `rg_cov`
Instruction Type	`branches`, `cmps`, `loads`, `binaryops`
Operands	`array_al`, `array_vector_al`, `struct_al`, `i8_arg`, `i16_arg`, `i64_arg`, `pointer_arg`, `pointer_st`
Comparison Type	`floats_cmps`, `pointer_cmps`, `i64_cmps`, `str_mem_cmps`
APIs	`m_ap`, `h_ap`
CFG	`min_path`, `avg_min_path`, `cycle`
DDG	`edge_ddgs`

Section 6.3, and Kendall's τ is computed similarly. For now, we will emphasize one important key difference between these correlation numbers. It is important to note that Spearman's R and Kendall's τ are relative, whereas Pearson's R is not. The Pearson's R that we compute here is the correlation of the Cohen's d to the baseline *naive* fuzzer, so this number involves only two fuzzers; the single fuzzer we are examining and the *naive* fuzzer. On the other hand, for the two rank correlations of Spearman's R and Kendall's τ, these two are the correlation of the ranks, and therefore, this number is relative to all the other fuzzers.

6.2.1 Schedulers. Table 3 summarizes the features correlated with the five different scheduler techniques we implemented. At first glance, we can see that the performance of all techniques, except for the random scheduler, is affected by several features of the target program. The highest Pearson's correlation, with the lower p-value, is with the average size of the initial seeds. This correlation is positive (always above 0.7) for the *explore*, *fast*, and *weighted* schedulers.

Similarly, the results also show that the AFLfast schedulers (*explore* and *fast*) provide better results on large programs (correlation with `bbs` above 0.5) while they might be negatively affected by the presence of many arrays (the conditional is due to p-values close to the 0.05 threshold). *Weighted* are also positively correlated with `br_cov`, `fn_cov`, `ln_cov`, `rg_cov`. These features are higher if the program already covers large program space even with the initial seeds. The results indicate that these schedulers work effectively if the fuzzer has lots of corpus entries to pick, and larger program space to explore. If we analyze the rank correlation for `i8_arg` and `i16_arg`, the values are not high. Therefore correlation of *fast* and *explore* to these features is not likely to be well-founded.

Table 3: Correlation of scheduler fuzzers

Feature	Pearson's R	P-value	Spearman's R	Kendall's τ
explore				
`array_al`	−0.440	0.046	−0.325	−0.215
`bbs`	0.567	0.006	0.319	0.219
`corpus_sz`	**0.724**	0.000	0.268	0.192
`floats_cmps`	0.499	0.030	0.228	0.170
`i16_arg`	0.448	0.047	0.119	0.0859
`i8_arg`	0.466	0.038	0.0877	0.0533
`ln_cov`	0.517	0.016	0.114	0.0741
fast				
`array_al`	−0.492	0.024	−0.368	−0.245
`array_vector_al`	−0.482	0.027	−0.363	−0.241
`bbs`	0.594	0.004	0.342	0.233
`corpus_sz`	**0.725**	0.000	0.224	0.154
`floats_cmps`	0.501	0.029	0.281	0.205
`i8_arg`	0.488	0.029	0.118	0.074
`ln_cov`	0.522	0.015	0.061	0.046
rand-scheduler				
`m_ap`	0.514	0.014	0.289	0.192
cov-accounting				
`array_al`	0.560	0.008	0.149	0.112
`array_vector_al`	0.560	0.008	0.150	0.112
`corpus_sz`	**-0.692**	0.001	−0.335	−0.224
`m_ap`	0.558	0.007	0.407	0.276
weighted				
`br_cov`	0.496	0.026	0.301	0.193
`corpus_sz`	**0.707**	0.000	0.371	0.259
`corpus`	0.515	0.029	0.388	0.277
`floats_cmps`	0.576	0.010	0.223	0.159
`fn_cov`	0.449	0.036	0.263	0.179
`ln_cov`	0.583	0.006	0.270	0.180
`rg_cov`	0.504	0.024	0.310	0.205
`h_ap`	−0.559	0.007	−0.211	−0.165

For the *cov-accounting*, we can observe a positive correlation with `m_ap`, which is a feature for the security-sensitive APIs. This association is logical since *cov-accounting* is specifically designed to give more importance to the code containing security-sensitive API calls.

> Schedulers such as *explore*, *fast*, *weighted* work well with larger programs, larger input, and higher initial coverage, and therefore, an increased number of testcases to schedule. The performance of these schedulers is influenced by how many corpus entries the scheduler has to schedule. The *cov-accounting* correlates positively with the security-sensitive API usage, which is expected from its design.

Table 4: Correlation of feedback fuzzers

Feature	Pearson's R	P-value	Spearman's R	Kendall's τ
ngram4				
avg_min_path	0.501	0.029	0.447	0.308
bbs	−0.575	0.006	−0.409	−0.300
corpus_sz	**-0.724**	0.000	−0.030	−0.029
min_path	0.522	0.022	0.456	0.308
ngram8				
bbs	−0.466	0.044	−0.533	−0.376
corpus_sz	−0.580	0.012	0.035	0.006
pointer_st	−0.484	0.036	−0.212	−0.114
str_mem_cmps	−0.537	0.018	−0.285	−0.157
value_profile				
branches	0.441	0.035	0.284	0.190
cycle	−0.464	0.026	−0.412	−0.288
edge_ddgs	0.488	0.025	0.343	0.250
i64_arg	0.434	0.043	0.270	0.190
i64_cmps	0.426	0.048	0.340	0.219
loads	0.579	0.006	0.316	0.223
pointer_arg	−0.465	0.025	−0.182	−0.124
naive-ctx				
corpus_sz	−0.649	0.004	0.017	0.0012
corpus	−0.526	0.030	−0.206	−0.143
floats_cmps	−0.507	0.038	−0.160	−0.121
pointer_cmps	−0.617	0.005	−0.443	−0.320
str_mem_cmps	**-0.733**	0.000	−0.392	−0.278

Table 5: Correlation of mutator fuzzers

Feature	Pearson's R	P-value	Spearman's R	Kendall's τ
mopt				
binaryops	−0.425	0.049	−0.031	−0.020
cmplog				
br_cov	−0.468	0.037	−0.343	−0.237
rg_cov	−0.504	0.024	−0.306	−0.211
grimoire				
corpus_sz	−0.622	0.003	−0.555	−0.397
ln_cov	−0.461	0.036	−0.367	−0.268
struct_al	**-0.687**	0.001	0.040	0.033
str_mem_cmps	0.428	0.047	0.298	0.206

6.2.2 Feedbacks. Table 4 summarizes the relationship between feedback fuzzers and program features. The first observation we can draw from these results is that the performance of fuzzer built with the *ngram4*, *ngram8*, and *naive-ctx* feedbacks is negatively correlated with features that indicate more seed corpus entries and larger program size. As in *ngram4* and *ngram8* negatively correlate with `bbs` and *naive-ctx* negatively correlates with `corpus`.

These three strategies reward the fuzzer with more complex coverage than mere edge coverage. For instance, the *ngram* fuzzers are better at exploring path coverage, and the *naive-ctx* fuzzer is for distinguishing different caller stacks. However, our result shows that as the number of corpus increases, the effectiveness of these techniques declines. This result is aligned with the conclusion from the previous research [45], which explains that more sensitive feedback could overwhelm the fuzzer corpus and degrade the fuzzer performance as it will be hard for the scheduler to schedule the testcases.

While `corpus` has a solid negative correlation with *naive-ctx*, the similar correlation of `corpus_sz` seems negatively strong, but the rank correlation suggests a much weaker correlation. In fact, the former correlation makes sense, but the latter correlation does not since the file size of the seed corpus entry does not suggest the increased number of testcases during fuzzing runs.

In contrast, `avg_min_path` and `min_path` are correlated with *ngram4*. These two features represent how many basic blocks the program path traverses in each function. As these numbers increase, the control flow gets more complex, and more paths exist to explore within each function. This result seems to support the fact that *ngram4*-based fuzzers successfully identify more paths, compared to the branch and edge coverage feedback, as they are designed to do.

The *value-profile* fuzzer seems to be best suited for fuzzing programs with more `branches` and `i64_cmps` since they relate to the frequencies of comparisons. The `i64_arg` feature correlates indirectly by positively correlating with `i64_cmps`. The mechanism of *value-profile* provides an explanation also for the positive correlation with `edge_ddgs` and `loads`, as a greater range of values flowing into the cmp instructions will be rewarded with unseen comparison arguments for the value profile.

Finally, we observe strong negative correlations between *naive-ctx* fuzzers and comparison-related features. While *naive-ctx* is not equipped with any technique to specifically overcome complex comparisons, the negative correlations are unexpectedly strong.

However, this can be interpreted by the inter-correlation between features. In fact, these comparison features that correlate to *naive-ctx* are somewhat positively inter-correlated to `bbs`, which, in turn, means that the fuzzer tends to have much more corpus testcases later on as the program space to explore grows. (For example, the correlation between `bbs` and `floats_cmps` is 0.27.) As a result, this trend leads to worsened performance. The critical observation is that there is only **correlation**, not **causation** between these two. In other words, here, the frequency of string and memory comparison did **not cause** the worse performance of *naive-ctx*. Instead, `str_mem_cmps` indirectly correlated negatively to *naive-ctx* by *bbs*, namely, the number of basic blocks.

> Complex feedback such as *ngram4*, *ngram8*, and *naive-ctx* will perform worse as the program and the input number increases. In contrast, some ngram fuzzers correlate positively with path complexity, and *value-profile* correlates positively with some comparison features.

6.2.3 Mutators. The last correlation group is from mutator fuzzers shown in Table 5. The first surprising result is that, although *cmplog* fuzzer implements a mutator that replaces the input with the observed value in the cmp instructions operand, we could not find any correlation with comparison-related features, such as the frequency of comparisons. Conversely, *cmplog* fuzzer correlates negatively with `br_cov` and `rg_cov`, suggesting that it is most effective when it starts with lower initial coverage values, requiring the fuzzer to explore larger code regions.

Another fuzzer, *grimoire*, is specialized in fuzzing programs that receive structured inputs. Correlations can be observed between features related to comparison, such as `str_mem_cmps`. These correlations are logical. Furthermore, we have noted a negative correlation between *grimoire* and the values of `corpus_sz` and `ln_cov`. This correlation can be explained using the same logic as used for

cmplog fuzzer. The more fuzzer-blockers the fuzzer has to break through in the initial state, the more influential the *cmplog* technique is. It is to our surprise that *cmplog* and *grimoire* did not show a strong positive correlation with the comparison related features and instead correlates (negatively) with the initial coverage.

Our hypothesis on this result is that this mutation technique can solve the comparisons in the program. However, we assume that compared to the number of all the comparisons in the program, the number of the comparisons that can be both 1. solved by these mutation techniques, and 2. cannot be solved by naive mutations is relatively small. This implies that the amount of comparisons alone cannot be a good indicator of the performance of *cmplog* and *grimoire*.

Meanwhile, the initial coverage has a strong inter-correlation with the corpus's testcase size. For example, `LN_COV` has a correlation value of 0.62 and 0.22 with `corpus_sz` and `corpus` respectively, meaning that higher initial coverage usually means the enriched corpus. In this case, the fuzzer already has a hint in the corpus on how to solve the comparisons and magic bytes, making the two fuzzing technique *cmplog* and *grimoire* less effective.

However, the correlation observed for `struct_al` is counter-intuitive, and furthermore, the rank correlation result does not agree. We suspect this strange correlation might be due to the influence of outliers, and we will discuss this issue in 8.5.

> Surprisingly, *cmplog* and *grimoire* do not correlate strongly with comparison-related features. Instead, they correlate with the initial state of fuzzing. The higher the initial coverage is, the more likely that their performance cannot show its edges.

6.3 Correlation Analysis: Rank

The correlation results presented so far were computed separately for each fuzzer and allowed us to understand how the effectiveness of each technique is affected by the different program features. We now repeat the analysis by using Spearman's correlation this time in order to study the relative performance of each fuzzer when compared with the others. This time, the correlation is computed on 460(=23×20) data points, each of them representing one trial of the fuzzer.

Each data point representing a trial is characterized by two values: the rank of the program's feature and the rank of the fuzzer's performance. For instance, if the first rank value is the 100th and the second rank value is the 200th, this means that this benchmark program used in this program has the 100th feature value among other benchmarks, and the fuzzer used in this trial has the 200th fuzzer performance among other fuzzers. Therefore, we find the correlation between these two rank values for the 460 data points. We first examine the ranking of the 13 fuzzers we developed (the baseline plus the 12 individual techniques) and then repeat the experiment with the three off-the-shelf fuzzers: AFL++, LibFuzzer, and HonggFuzz. This allows us to directly compare our results with the ones reported in [47].

Again, for the sake of clarity, we only report results where Spearman's R is greater than or equal to 0.40, or less than or equal to -0.40, and similarly, we also present the Kendall's τ for reference.

Table 6: Spearman correlation of feedback fuzzers. The P-value is near 0 for all the variants and thus omitted.

Feature	Spearman's R	Kendall's τ
ngram4		
`avg_min_path`	0.447	0.308
`bbs`	−0.404	−0.265
`i16_arg`	−0.409	−0.300
`min_path`	0.456	0.308
ngram8		
`bbs`	−0.533	−0.376
`size`	−0.479	−0.336
naive-ctx		
`bbs`	−0.415	−0.297
`pointer_cmps`	−0.443	−0.320
grimoire		
`corpus_sz`	−0.555	−0.397
value-profile		
`cycle`	−0.412	−0.288
`ln_cov`	−0.431	−0.298
`rg_cov`	−0.417	−0.296
`m_ap`	0.407	0.276

6.3.1 Rank Correlations of LibAFL Fuzzers. Table 6 shows the Spearman's correlation for the 13 LibAFL fuzzers. For this case, only 6 of the 12 techniques we analyzed show a moderate correlation wrt program features. In all the results, p-values are close to zero; this is because we used many more data points (460) than Pearson's correlation. With a larger number of data, the correlation becomes more statistically significant. In general, the result of Spearman's R is quite similar to Kendall's τ. The findings are fewer but in line with those we discovered by using Cohen's d. For instance, Spearman correlation confirms that all *ngram*-based solutions and the *naive_ctx* fuzzer are negatively affected by the program sizes (`bbs` and `size`). They also confirm that *grimoire* tends to perform worse than other fuzzer with large corpus's testcase sizes, while *cov-accounting* performs better in programs that invoke many security-related APIs.

Table 7 shows the results for AFL++, LibFuzzer, and HonggFuzz. For the purpose of comparing our results with those of Wolff et al. [47], we consider the same features that they use. Likewise, the result of Kendall's τ is similar to Spearman's R but with a less strong value. Interestingly, some features positively affect AFL++ and have a negative effect on LibFuzzer (since this is a ranking correlation, whenever something benefits the ranking of a fuzzer it also has a negative effect on the ranking of others). In particular, this is the case for the size and number of the initial corpus (which also helps HonggFuzz) as well as for metrics related to the initial coverage.

These findings are in line with the ones of Wolff et al., who reported that "*LibFuzzer's ranking worsens as the coverage, average size and average execution time of the seeds in the initial corpus increases, or as the size of the program increases. [...] Similarly,* AFL++ *'s ranking improves on benchmarks with larger programs or where the initial seed corpus executes faster.*"

Table 7: Comparison of AFL++, LibFuzzer, and honggfuzz

Feature	Spearman's R	Kendall's τ
aflplusplus		
corpus	0.163	0.119
corpus_sz	0.238	0.163
ln_cov	0.301	0.213
fn_cov	0.307	0.209
cmps	0.333	0.233
br_cov	0.336	0.228
rg_cov	0.362	0.247
honggfuzz		
cmps	−0.161	−0.108
fn_cov	−0.104	−0.0513
corpus	0.107	0.0620
corpus_sz	0.194	0.141
size	0.252	0.174
bbs	0.313	0.209
libfuzzer		
rg_cov	−0.321	−0.213
br_cov	−0.316	−0.207
ln_cov	−0.299	−0.195
corpus	−0.246	−0.177
corpus_sz	−0.232	−0.164
fn_cov	−0.171	−0.116
bbs	−0.119	−0.0757

The rank correlation analysis on LibAFL-based fuzzers shows a similar trend as Pearson's correlation analysis. Meanwhile, our comparison on AFL++, LibFuzzer, and HonggFuzz also aligns with the result from the previous study.

7 Prediction Results

The correlation results we presented in the previous section provide a valuable guideline on how to choose different algorithms to increase the accuracy of a fuzzing campaign. However, features are not mutually exclusive, and therefore, a given target might have two different characteristics that favor different techniques. In that case, it is unclear which choice would be more beneficial simply by looking at the correlation results.

For this reason, as explained in Section 3, we divided the techniques into three groups (schedulers, feedbacks, and mutators), and then trained a separate ML predictor to rank all the methods from each group. Table 8 summarizes the results, showing for each target which one was the technique suggested by our classifier, its rank in the actual experiments, and the technique that provided the best results in practice. We would like to stress that our prediction model was trained on different programs, and therefore, without any previous knowledge about the actual experiment results.

The first important observation is that there is no technique that is universally superior per se. In fact, all six schedulers, four feedback approaches, and three mutators returned the (actual) best results on at least one target. This supports our initial hypothesis that simply adopting one technique is rarely the best approach, and thus, being able to predict which fuzzing solution to use for a given experiment will play a key role in the future of software testing.

In 42.4% of the cases, our prediction model correctly chose the best-performing technique, and in 67% of the experiments, the predicted choice was in the top-two best performers. It is important to note that in some categories, the best technique and the second best performed equally well, with a difference in the effect size of less than 0.2, which is considered to be a small effect size, according to Sullivan. et al [43]. These cases are marked with an asterisk $*$ sign in the table. Table 9 compares the actual branch coverage obtained by the baseline fuzzer, the fuzzer obtained by our prediction classifiers, AFL++, LibFuzzer, and HonggFuzz . [2]

Each cell in the table reports the raw branch coverage as well as its improvement ratio compared to the baseline fuzzer. Overall, our predicted solution and LibFuzzer ranked first on four benchmarks each, AFL++ dominated on three, and HonggFuzz on none. More importantly, LibFuzzer discovered on average 1% more coverage than the baseline fuzzer, followed by HonggFuzz with 3% and AFL++ with 5%. The fuzzer based on our prediction resulted instead in a 12% improvement over the baseline – more than the other three off-the-shelf fuzzers combined.

8 Discussion

In this section, we interpret our results, compare them against previous research, and discuss alternative design choices and the potential limitations of our approach.

8.1 Novelty

Our research deviates from previous studies in two fundamental directions. First, we broke down fuzzing into its basic components and compared them against a baseline implementation, instead of comparing pre-assembled fuzzers made of fixed components.

Second, we compare fuzzers and compute which aspects of the target programs affect given techniques by using effect size rather than comparing the ranking of pre-selected fuzzers.

The issue with comparing pre-assembled fuzzers such as AFL++, LibFuzzer, and HonggFuzz is that it does not allow us to distinguish the root cause of the difference in the fuzzer performance. For instance, since AFL++ employs the *FAST* power schedule, a standard branch coverage feedback, and the *cmplog* mutator, it is hard to separate the contribution of the three and to tell which advantages or disadvantages each component brings to a certain result. If AFL++ performs better than LibFuzzer, this might be due just to its scheduler or its set of mutators. Unfortunately, we are unable to find the source of the difference. Our approach allowed us instead to perform a more fine-grained attribution and precisely pinpoint how the choice of each specific fuzzer component correlated with the final results.

Next, we investigated the limitations of rank comparisons. The critical aspect is that the correlation analysis for ranking will only explain how one fuzzer performs relative to others. This means that comparing different fuzzers would, therefore, often return different results and that the entire analysis needs to be repeated when a new fuzzer is added to the arsenal. In other words, this result is only valid as long as the set of fuzzers used remains the same.

[2] On some of the benchmarks, AFL++, LibFuzzer, or HonggFuzz did not build denoted by "–" in the table.

Table 8: The fuzzer ranks of the prediction and experiment result

Benchmark name	Technique group	Predicted technique	Predictions' rank	The best technique
assimp_assimp_fuzzer	schedulers	*fast*	1/6	*fast*
	feedbacks	*naive*	2/5	*value-profile*
	mutators	*cmplog*	1/3	*cmplog*
astc-encoder_fuzz_astc_physical_to_symbolic	schedulers	*explore*	2/6	*weighted*
	feedbacks	*value-profile*	1/5	*value-profile*
	mutators	*mopt*	1/3	*mopt*
brotli_decode_fuzzer	schedulers	*fast*	3/6	*weighted*
	feedbacks	*value-profile*	5/5	*naive*
	mutators	*naive*	2/3*	*mopt*
double-conversion_string_to_double_fuzzer	schedulers	*fast*	3/6	*cov-accounting*
	feedbacks	*ngram8*	2/5*	*value-profile*
	mutators	*cmplog*	1/3	*cmplog*
draco_draco_pc_decoder_fuzzer	schedulers	*fast*	1/6	*fast*
	feedbacks	*value-profile*	1/5	*value-profile*
	mutators	*cmplog*	1/3	*cmplog*
fmt_chrono-duration-fuzzer	schedulers	*fast*	4/6	*rand-scheduler*
	feedbacks	*value-profile*	1/5	*value-profile*
	mutators	*cmplog*	3/3	*naive*
guetzli_guetzli_fuzzer	schedulers	*explore*	4/6	*weighted*
	feedbacks	*value-profile*	1/5	*value-profile*
	mutators	*naive*	2/3*	*mopt*
icu_unicode_string_codepage_create_fuzzer	schedulers	*explore*	3/6	*weighted*
	feedbacks	*value-profile*	3/5	*naive*
	mutators	*cmplog*	3/3	*naive*
libaom_av1_dec_fuzzer	schedulers	*explore*	2/6	*weighted*
	feedbacks	*value-profile*	1/5	*value-profile*
	mutators	*mopt*	1/3	*mopt*
libcoap_pdu_parse_fuzzer	schedulers	*rand-scheduler*	6/6	*naive*
	feedbacks	*value-profile*	2/5	*ngram8*
	mutators	*cmplog*	1/3	*cmplog*
libhevc_hevc_dec_fuzzer	schedulers	*explore*	1/6	*explore*
	feedbacks	*value-profile*	2/5	*ngram4*
	mutators	*cmplog*	3/3	*naive*

Table 9: The branch coverage result of SBFT benchmark programs

Benchmark name	Baseline	Prediction best	AFLplusplus	LibFuzzer	Honggfuzz
assimp_assimp_fuzzer	2849.0	5755.5(202%)	3366.0(118%)	2923.5(103%)	–
astc-encoder_fuzz$_a$stc_physical_to_symbolic	489.0	492.0(101%)	–	511.0(104%)	–
brotli_decode_fuzzer	902.0	903.5(100%)	903.0(100%)	–	–
double-conversion_string_to_double_fuzzer	498.0	495.0(99%)	507.5(102%)	–	502.0(101%)
draco_draco_pc_decoder_fuzzer	1514.0	1924.0(127%)	1796.5(119%)	–	1731.5(114%)
fmt_chrono-duration-fuzzer	1082.0	1086.0(100%)	1088.0(101%)	1094.0(101%)	1091.0(101%)
guetzli_guetzli_fuzzer	1492.0	1492.5(100%)	1497.0(100%)	1456.0(98%)	1470.5(99%)
icu_unicode_string_codepage_create_fuzzer	1339.0	1302.0(97%)	1340.0(100%)	1341.0(100%)	1316.5(98%)
libaom_av1_dec_fuzzer	10968.5	11235.5(102%)	11084.0(101%)	10171.0(93%)	11016.0(100%)
libcoap_pdu_parse_fuzzer	745.0	758.0(102%)	817.0(110%)	825.0(111%)	824.0(111%)
libhevc_hevc_dec_fuzzer	10321.5	10334.0(100%)	10353.0(100%)	10269.0(99%)	10321.5(100%)
Mean improve over the baseline		**112 %**	105%	101%	103%

Table 10: Unstability of rank correlations

Target	Fuzzer A	Fuzzer B	Fuzzer C
T_1	2	5	8
T_2	5	2	1

We can demonstrate this concept with a simple example. Let's say that we have fuzzer A and fuzzer B with the branch coverage score shown in the Table 10. Based on these results, fuzzer A ranks second on program T_1 and first on program T_2, while fuzzer B ranks first on program T_2 and second on T_1. By observing only these two fuzzers, we could claim that fuzzer B positively correlates with some features in program T_1 since its ranking is better on that target.

However, if we add fuzzer C into the comparison, it completely alters the conclusion and even the relationship among the other two fuzzers. Now, the B ranks second on both targets, T_1 *and* T_2. Therefore, if we only consider the ranking for comparison, the positive correlation we previously observed now disappears. The rankings were affected by a change in the set of fuzzers, leading to a subsequent impact on the correlation and the overall conclusions of the experiment.

This form of **instability** makes the results difficult to generalize, and is an inevitable drawback of this approach. Meanwhile, our approach based on the effect size remains unaffected by changes in the set of fuzzers under comparisons, as the correlation between a technique and a set of features of the program is not affected by other fuzzers. This makes our solution a more reliable method for evaluating correlations.

One potential downside of our approach is that Pearson's correlation may not be the most appropriate metric if the dataset does not exhibit a linear relationship. We checked this by plotting the data and did not observe any obvious curvy relationships. However, it is not possible to determine what relationship exists between the program feature and fuzzer performances beforehand. In fact, determining this relationship is the very objective of this work. This is why we have also included the results of Spearman's correlation and Kendall's tau in our evaluation for a more comprehensive evaluation to supplement our analysis and address this limitation.

8.2 Classification Models

Wolff et al. employed a multiple linear regression model to quantify the effect of benchmarks and program properties. Therefore, it might have seemed natural for us to employ a similar MLR (multiple linear regression) model as well. However, multicollinearity, namely the correlation between the predictors [18], introduced an obstacle to the application of a multiple linear regression model to our data. Previous research was not affected by this problem since the number of features used for their MLR models were small.

In our case, instead, we extracted 59 program features that might or might not correlate with each other. For example, the frequency of `cmp` instruction and the frequency of `i32 cmp` instruction naturally have positive correlation with each other. We detailed and diversified the program features to have a finer analysis of the program; however, the increased complexity made it impossible for us to apply a liner regression model, and we decided to make use of random forest model to circumvent the problem.

8.3 Alternative Coverage metrics

We used branch coverage for our evaluation. In fact, evaluating the fuzzer performance based on branch coverage can be considered the golden standard, as branch or edge coverage, is the most used evaluation metric in recently published papers [37]. Besides the standard branch coverage, we also considered function coverage metrics. However, the result provides less insightful results, and therefore, we published the result on our artifact page.

8.4 Interference

One of the potential drawbacks of our approach is that we assume that fuzzing techniques are orthogonal and independent. For instance, we separately predict the feedback component and the scheduler. However, in reality, the combination of two techniques might have consequences that could positively or negatively affect the performance (i.e., a given scheduler might undermine the advantages brought by the feedback algorithm).

Ideally, one should run the experiment with every possible combination of techniques. However, this would require an unfeasible amount of time and resources, as we need to try every combination of different techniques from each category. Instead, we conducted a small-scale experiment to assess the degree of the interference. Suppose that we have two techniques, t_1, t_2. We can define their effect on the fuzzer performance compared to the baseline fuzzer as Δt_1 and Δt_2. If we consider the interaction between techniques, then the predicted performance is modeled as

$$perf = baseline + \Delta t_1 + \Delta t_2 + \Delta t_1 \Delta t_2$$

We can compare the degree of the "non-linear" term $\Delta t_1 \Delta t_2$ to assess the degree of interference.

The most obvious possible interference is between the scheduler and the feedback. In [45], it was concluded that "*A more sensitive coverage metric may select more inputs as seeds, but the fuzzer may not have enough time budget to schedule all the seeds or mutate them sufficiently.*" Therefore, in our experiment, we will examine the interference of the combination of the scheduler technique and feedback technique. To this end, we will evaluate the performance of fuzzer pairs of (*fast*, *ngram4*), and (*fast*, *value-profile*) using the same 22 benchmarks we used to compute the correlation. [3].

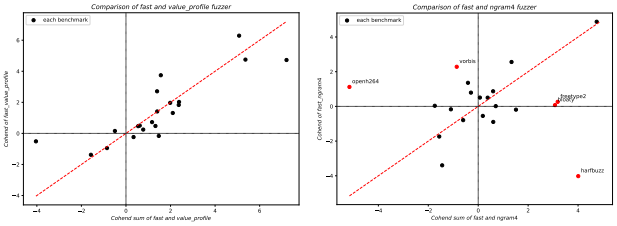
First, we evaluate the performance of the individual fuzzer of the pairs by measuring its effect size, namely Cohen's d, compared to the baseline fuzzer. Next, we measure the performance of the single fuzzer using both techniques combined. If the sum of the individual performance from the two technique pairs is similar to the performance of the combined fuzzer, then we can say that the effect of the interference or the non-linear term is small. The result is presented in Figure 4.

Each dot represents an individual benchmark, and the x-axis value represents the sum of Cohen's d of the two fuzzers in the same pair, and the y-axis value represents the Cohen's d of the combined fuzzer.

The closer the scattered data points are to the reference line "y = x", the smaller the non-linear term is; therefore, the interference between the two techniques is minor. For the technique pair of (*fast*,

[3] For one of the benchmark out of 23, *libpcap_fuzz_both*, we could not retrieve the result

Figure 4: Interference assessment



value_profile), almost all the data points are pretty close to the reference line, and therefore we can conclude that for this technique, their contributions are orthogonal and do not interfere too much.

For the technique pair of (*fast*, *ngram4*), we see some remote data points colored in red. Interestingly, the three data points in the bottom right, *bloaty*, *harfbuzz*, and *freetype2* are all large programs. Furthermore, the fact that these points are below the reference line means that the combined fuzzer is worse than the sum of *fast* and *ngram4*. This observation exactly matches with the previous statement from [45], as with *ngram4* feedback, when the target gets large, the *fast* scheduler has difficulty scheduling the testcases, resulting in worse performance.

While our experiment provides some insight into the potential interference between techniques, such as the pair of *fast* and *value_profile* showing little interference, and the pair of *fast* and *ngram4* exhibiting noticeable interference in several cases, particularly for larger benchmarks. It is important to note that the scale of our experiment is limited, and further research with more extensive data is needed to draw definitive conclusions.

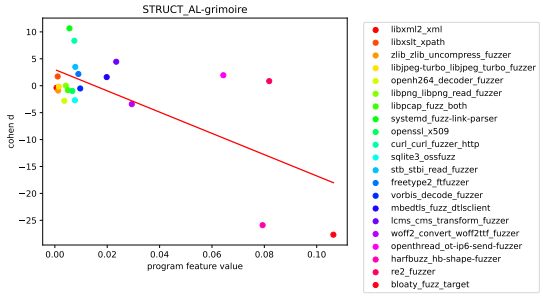
8.5 Limitations and Future Works

We conclude the discussion section with an overview of our study's limitations and future works. First of all, it is essential to clarify that our research centers on **branch coverage** and identify program features that correlate with this metric. We then proceed to create prediction models that select the combination of fuzzing techniques that output the highest **branch coverage**.

However, there is a difference between coverage and the ability to find bugs. Böhme's research [14] highlighted the strong correlation between high coverages and the ability to discover bugs in modern fuzzers, although the study also claimed that there is no firm agreement between the two metrics. As a result, it is important to stress that our study focuses solely on coverage and does not make any claims about the actual bug-finding ability of the different fuzzers.

One line of possible improvement is to extend the benchmark. Diversifying the benchmark suite would help to limit the influence of outliers, which in a few cases introduced spurious correlations in our results. For instance, Figure 5 shows such a case. The image depicts the correlation between the *grimoire* fuzzer and `struct_al` program feature. As we reviewed in Section 6, this produced a counter-intuitive result. Focusing on the graph, we observe that the majority of the program feature values are locally concentrated around one side, causing the entire linear relationship to lean towards the outliers. In our experiment, we employed 34 (23 + 11) benchmark programs, and this dataset already required 250,240

Figure 5: An example correlation affected by the outliers



cpu hours to test them on 16 fuzzer, and it was hard to extend the benchmark suite due to the limitation of the resource.

9 Related Works

One of our challenges is to identify the optimal fuzzer configuration from a vast array of techniques, and collaborative fuzzing - a method of coordinating multiple different fuzzers during testing - is a related area of interest. Enfuzz [15] and Collabfuzz [35] are the initial attempts to coordinate various types of fuzzers to enhance the results. The motivation behind these works is that when different fuzzers are combined and run together, the program space missed by one fuzzer can be covered and complemented by others. The main challenge of collaborative fuzzing is to identify the most effective combination of fuzzers to use.

However, the ultimate goal of these collaborative fuzzing frameworks and our work differs. What collaborative fuzzing answers is the combination of pre-made fuzzers complementing each other when they are run together. On the other hand, our work aims to give the best choice of techniques to fuzz when you use a single fuzzer to fuzz. This is not what collaborative fuzzing answers for us.

Additionally, we reiterate that the key differences that distinguishes our study from prior research is that prior studies employed fuzzers that do not share the same baseline. The fact that using fuzzers that cannot be decomposed into orthogonal approaches will mistify the individual contribution of its techniques.

10 Conclusion

We presented a novel approach to correlate static analysis features extracted during compilation with the performance results of various fuzzing techniques and, on top of this, we developed a prediction model to get a combination of techniques to be used as fuzzer configuration tailored for each target.

Our research pioneers the investigation of the relationship between program characteristics and fuzzer performance. We hope that our work will serve as the foundational work in this field.

Ackoledgements. We want to thank the anonymous reviewers, Slasti Mormanti for his valuable insights and the LibAFL community.

References

[1] 2020. LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html. [Online; accessed July 15, 2024].

[2] Accessed July 15, 2024. Binutils - GNU Project - Free Software Foundation. https://www.gnu.org/software/binutils/.

[3] Accessed July 15, 2024. Definition of DDG in the LLVM framework. https://llvm.org/docs/DependenceGraphs/index.html.

[4] Accessed July 15, 2024. llvm-cov - emit coverage information. https://llvm.org/docs/CommandGuide/llvm-cov.html.

[5] Accessed July 15, 2024. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[6] Accessed July 15, 2024. Pooled Standard Deviation | Wolfram Formula Repository. https://online.stat.psu.edu/stat500/lesson/7/7.3/7.3.1/7.3.1.1. [Online; accessed July 15, 2024].

[7] Frances E. Allen. 1970. Control Flow Analysis. *SIGPLAN Not.* 5, 7 (jul 1970), 1–19. https://doi.org/10.1145/390013.808479

[8] Cornelius Aschermann, Tommaso Frassetto, T. Holz, Patrick Jauernig, A. Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*.

[9] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*.

[10] Gérard Biau and Erwan Scornet. 2016. A random forest guided tour. *Test* 25 (2016), 197–227.

[11] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1985–2002.

[12] Marcel Böhme, Valentin Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1–11.

[13] 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* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[14] 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 (ICSE '22)*. https://doi.org/10.1145/3510003.3510230

[15] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. {EnFuzz}: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1967–1983.

[16] Jacob Cohen. 1992. Statistical power analysis. *Current directions in psychological science* 1, 3 (1992), 98–101.

[17] Adele Cutler, D. Richard Cutler, and John R. Stevens. 2012. *Random Forests*. Springer New York, New York, NY, 157–175. https://doi.org/10.1007/978-1-4419-9326-7_5

[18] Jamal I Daoud. 2017. Multicollinearity and regression analysis. In *Journal of Physics: Conference Series*, Vol. 949. IOP Publishing, 012009.

[19] 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)*. USENIX Association.

[20] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. (2022).

[21] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. 2023. Dissecting American Fuzzy Lop: A FuzzBench Evaluation. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 52 (mar 2023), 26 pages. https://doi.org/10.1145/3580596

[22] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. 2023. autofz: Automated Fuzzer Composition at Runtime. In *Proceedings of the 32st USENIX Security Symposium (Security)*. Anaheim, CA.

[23] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. *Annual Computer Security Applications Conference* (2020).

[24] Jan Hauke and Tomasz Kossowski. 2011. Comparison of values of Pearson's and Spearman's correlation coefficients on the same sets of data. *Quaestiones geographicae* 30, 2 (2011), 87–93.

[25] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. [n. d.]. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. https://doi.org/10.1145/3460319.3464795

[26] Louis M Hsu and Ronald Field. 2003. Interrater agreement measures: Comments on Kappan, Cohen's Kappa, Scott's π, and Aickin's α. *Understanding Statistics* (2003).

[27] 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 (CCS '18)*. https://doi.org/10.1145/3243734.3243804

[28] D. Liu, J. Metzman, M. Bohme, O. Chang, and A. Arya. 2023. SBFT Tool Competition 2023 - Fuzzing Track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE Computer Society, Los Alamitos, CA, USA, 51–54. https://doi.org/10.1109/SBFT59156.2023.00016

[29] LLVM Project. [n. d.]. LibFuzzer - Value Profile. https://llvm.org/docs/LibFuzzer.html#value-profile. [Online; accessed July 15, 2024].

[30] LLVM Project. 2018. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. [Online; accessed July 15, 2024].

[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)*. USENIX Association, Santa Clara, CA, 1949–1966.

[32] V. Manes, H. Han, C. Han, S.K. Cha, M. Egele, E. J. Schwartz, and M. Woo. 5555. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 01 (oct 5555). https://doi.org/10.1109/TSE.2019.2946563

[33] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with data dependency information. In *EuroS&P 2022, 7th IEEE European Symposium on Security and Privacy, 6-10 June 2022, Genoa, Italy*, IEEE (Ed.). Genoa.

[34] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin 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*. New York, NY, USA.

[35] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. CollabFuzz: A Framework for Collaborative Fuzzing. In *Proceedings of the 14th European Workshop on Systems Security* (Online, United Kingdom) *(EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3447852.3458720

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[37] 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 Computer Society, 137–137.

[38] Patrick Schober, Christa Boer, and Lothar A Schwarte. 2018. Correlation coefficients: appropriate use and interpretation. *Anesthesia & analgesia* 126, 5 (2018), 1763–1768.

[39] Pranab Kumar Sen. 1968. Estimates of the regression coefficient based on Kendall's tau. *Journal of the American statistical association* 63, 324 (1968), 1379–1389.

[40] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[41] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[42] Prashast Srivastava and Mathias Payer. [n. d.]. Gramatron: Effective Grammar-Aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. https://doi.org/10.1145/3460319.3464814

[43] Gail M Sullivan and Richard Feinn. 2012. Using effect size—or why the P value is not enough. *Journal of graduate medical education* 4, 3 (2012), 279–282.

[44] Robert Swiecki. [n. d.]. Honggfuzz. https://github.com/google/honggfuzz. [Online; accessed July 15, 2024].

[45] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 1–15.

[46] 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*.

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

[48] Michał Zalewski. 2016. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt. [Online; accessed July 15, 2024].

A Program features summary

In Table 11, we summarize the program features utilized in our experiments, and indicate whether we observed correlation or not.

Table 11: All the extracted program features.
Features in blue correlates with at least one fuzzer, while the ones in red don't have any correlation.

Feature name	Type	Interpretation
bbs	Generic Feature	Number of basic blocks
size	Generic Feature	Size of the program binary in bytes
ne_level	Generic Feature	Nested level of the source code program
corpus	Initial Seeds	Number of initial corpus entry
corpus_sz	Initial Seeds	Size of initial corpus entry
br_cov	Initial Seeds	Initial branch coverage
ln_cov	Initial Seeds	Initial line coverage
fn_cov	Initial Seeds	Initial function coverage
rg_cov	Initial Seeds	Initial region coverage
branches	Instruction Type	Number of branch instructions per BB
cmps	Instruction Type	Number of cmp instructions per BB
loads	Instruction Type	Number of load instructions per BB
binaryops	Instruction Type	Number of binary op instructions per BB
alloca	Instruction Type	Number of alloca instructions per BB
call	Instruction Type	Number of call instructions per BB
stores	Instruction Type	Number of store instructions per BB
array_al	Operand Type	Number of array alloca instructions per BB
array_vector_al	Operand Type	Number of array and vector alloca instructions per BB
struct_al	Operand Type	Number of structure alloca instructions per BB
i8_arg	Operand Type	Number of i8 argument in call instruction per BB
i16_arg	Operand Type	Number of i16 argument in call instruction per BB
i64_arg	Operand Type	Number of i64 argument in call instruction per BB
pointer_arg	Operand Type	Number of pointer argument in call instruction per BB
pointer_st	Operand Type	Number of pointer store instruction per BB
i32_arg	Operand Type	Number of i32 argument in call instruction per BB
i32_i64_arg	Operand Type	Number of i32 and i64 argument in call instruction per BB
i32_al	Operand Type	Number of i32 alloca instruction per BB
i64_al	Operand Type	Number of i64 alloca instruction per BB
i32_i64_al	Operand Type	Number of i32 and i64 alloca instruction per BB
i32_i64_i128_al	Operand Type	Number of i32 and i64 and i128 alloca instruction per BB
pointer_al	Operand Type	Number of pointer alloca instructions per BB
i32_i64_i128_arg	Operand Type	Number of i32 and i64 and i128 argument in call instruction per BB
i8_st	Operand Type	Number of i8 store instruction per BB
i16_st	Operand Type	Number of i16 store instruction per BB
i32_st	Operand Type	Number of i32 store instruction per BB
i64_st	Operand Type	Number of i64 store instruction per BB
i32_i64_st	Operand Type	Number of i32 and i64 store instruction per BB
i32_i64_i128_st	Operand Type	Number of i32 and i64 and i128 store instruction per BB
pointer_st	Operand Type	Number of pointer store instruction per BB
floats_cmps	Comparison Type	Number of float comparisons per BB
pointer_cmps	Comparison Type	Number of pointer comparisons per BB
i64_cmps	Comparison Type	Number of i64 comparisons per BB
str_mem_cmps	Comparison Type	Number of string or memory comparison APIs per BB
vector_cmps	Comparison Type	Number of vector comparisons per BB
array_vector_cmps	Comparison Type	Number of array and vector comparisons per BB
int_cmps	Comparison Type	Number of integer comparisons per BB
i8_cmps	Comparison Type	Number of i8 comparisons per BB
i16_cmps	Comparison Type	Number of i16 comparisons per BB
i32_cmps	Comparison Type	Number of i32 comparisons per BB
i64_cmps	Comparison Type	Number of i64 comparisons per BB
i32_i64_cmps	Comparison Type	Number of i32 and i64 comparisons per BB
i32_i64_i128_cmps	Comparison Type	Number of i32 and i64 and i128 comparisons per BB
m_ap	API Type	Security-sensitive API calls per BB
h_ap	API Type	Heap memory API calls per BB
min_path	CFG Feature	The length of shortest paths per CFG (There are multiple shortest paths)
avg_min_path	CFG Feature	Average length of shortest paths per CFG
cycle	CFG Feature	Occurrence of cycles (loop-back edges) per CFG
edge_ddgs	DDG Feature	Number of edges per DDG
node_ddgs	DDG Feature	Number of nodes per DDG