Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties

DYLAN WOLFF, National University of Singapore, Singapore

MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany

ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

In a typical experimental design in fuzzing, we would run two or more fuzzers on an appropriate set of benchmark programs plus seed corpora and consider their ranking in terms of code coverage or bugs found as outcome. However, the specific characteristics of the benchmark setup clearly can have some impact on the benchmark outcome. If the programs were larger, or these initial seeds were chosen differently, the same fuzzers may be ranked differently; the benchmark outcome would change. In this paper, we explore two methodologies to *quantify the impact of the specific properties on the benchmarking outcome.* This allows us to report the benchmarking outcome counter-factually, e.g., "If the benchmark had larger programs, this fuzzer would outperform all others". Our first methodology is the *controlled experiment* to identify a causal relationship between a single property in isolation and the benchmarking outcome. The controlled experiment requires manually altering the fuzzer or system under test to vary that property while holding all other variables constant. By repeating this controlled experiment for multiple fuzzer implementations, we can gain detailed insights to the different effects this property has on various fuzzers. However, due to the large number of properties and the difficulty of realistically manipulating one property exactly, control may not always be practical or possible. Hence, our second methodology is *randomization* and non-parametric regression to identify the strength of the relationship between arbitrary benchmark properties (i.e., covariates) and outcome. Together, these two fundamental aspects of experimental design, *control* and *randomization*, can provide a comprehensive picture of the impact of various properties of the current benchmark on the fuzzer ranking. These analyses can be used to guide fuzzer developers towards areas of improvement in their tools and allow researchers to make more nuanced claims about fuzzer effectiveness. We instantiate each approach on a subset of properties suspected of impacting the relative effectiveness of fuzzers and quantify the effects of these properties on the evaluation outcome. In doing so, we identify multiple properties, such as the coverage of the initial seed-corpus and the program execution speed, which can have statistically significant effect on the *relative* effectiveness of fuzzers.

1 INTRODUCTION

Fuzzing [8] is a well-known automated software testing method for finding security flaws by generating invalid or unexpected inputs. In particular, greybox fuzzers, which leverage light-weight instrumentation feedback to guide test input generation, have emerged as one of the most successful automatic bug finding approaches in practice [28]. Fuzzing has also emerged as an important research topic, with over 50 fuzzing papers published in the "Big Four" academic computer security conferences in 2024 alone (i.e., CCS, NDSS, S&P, USENIX Security)!

Yet, which fuzzer performs best and when?

Authors' addresses: Dylan Wolff, wolffd@comp.nus.edu.sg, National University of Singapore, Singapore; Marcel Böhme, marcel.boehme@mpi-sp.org, Max Planck Institute for Security and Privacy, Germany; Abhik Roychoudhury, abhik@comp.nus.edu.sg, National University of Singapore, Singapore.

Recently, the fuzzing community has identified *sound fuzzer evaluation* as a Top-3 most important research challenge [8]. To demonstrate improvement over the state-of-the-art, many fuzzing related benchmarks have been introduced. For instance, the MAGMA benchmark [32] provides a set of 138 real bugs in 9 programs. ProFuzzbench [52] offers access to 11 protocol implementations for network-enabled fuzzers. The FuzzBench benchmark [49] offers access to over 650 open-source programs via an integration with the OSS-Fuzz project [2]. FuzzBench is officially developed by Google demonstrating a substantial practical interest in sound fuzzer evaluation. Using these benchmarks, two or more fuzzers are compared by ranking them in terms of their performance (e.g., coverage achieved or #bugs found) [7, 13, 41]. To ensure that the observed performance differences are not due to random effects, tool developers are encouraged to repeat the experiments at least twenty times and measure effect size and statistical significance [6, 41]. To ensure that the observed performance differences can be attributed precisely to the proposed improvements, tool developers are encouraged to compare the prototype to the baseline which was extended to implement the improvement. To ensure that the observed performance differences are general, benchmark designers attempt to select a sample of subjects that are representative of the population of systems. Within this sample, there can be wide variations in benchmark properties by virtue of it being composed of disparate representative programs.

We observe that the benchmark outcomes depend critically on the specific properties of the selected benchmark. *On the average*, most fuzzers perform similarly while for each *specific* program there are often clear winners. For instance, in a recent FuzzBench experiment involving 23 programs and 11 fuzzers, we can see that the average ranking for the majority of fuzzers is 5.5 ± 1[1] – most fuzzers rank approximately the same. Looking only at these overall rankings across all benchmarks, it is not apparent that e.g. the ranking of AFL++ improves on larger programs. We call this evaluation methodology *atomistic*, because it does not account for the effects of benchmark properties. In other words, the final outcome of such an *atomistic* evaluation is *specific* to the current choice of benchmark and provides no insights into the *conditions* under which one fuzzer performs better than another. We are not the first to make this observation; it is well known that some additional variables, i.e., *covariates*, can have a different relative impact on fuzzer effectiveness, and thus benchmarking outcomes [34]. However, while this knowledge of that a particular covariate can possibly impact a fuzzer is a step in the right direction, in many cases these results are not actionable. There is currently no guidance for how to account for these covariates in future evaluations or assess their possible interactions with other variables in the evaluation setup.

In this paper, we propose two methodologies, control and randomization, which provide an *actionable* framework to can account for the effects of covariates in fuzzer evaluations. Using this framework, we suggest to report the benchmarking outcome together with the *conditions under which the outcome would change*. The first component of our methodology, a controlled experiment where one benchmark property is manipulated while all others are kept constant, can establish the degree to which changing that property *causes* a change in the response variable. However, due to the large number of possible properties and the difficulty of realistically varying a property and exactly one property, the first methodology may not always be practical or possible. Hence, we propose a second component to our methodology based on randomization and non-parametric multiple regression to identify the strength of the relationship between arbitrary benchmark properties (i.e., covariates) and the benchmarking outcome. In doing so, one can effectively *subtract* the influence of covariates without controlling their values.

Our methodologies allow users to evaluate the degree to which the benchmarking outcome is influenced by differences of the fuzzers (which is the focus of the evaluation) *as a function* of the properties of the benchmark. For fuzzer developers,

---

[1]https://www.fuzzbench.com/reports/2022-04-19/index.html

we see our approach as a way to gain insights into the variables which influence the effectiveness of their tools. For fuzzer evaluators, we believe our framework provides the tools to make knowledge of important covariates *actionable* by *subtracting* their influence in realistic (uncontrolled) benchmarking scenarios.

Control. Our first methodology is to control for the effect of a specific benchmark property. We suggest to keep all other properties fixed while varying only the property of interest. In doing so, we can directly identify the contribution of that property to the benchmarking outcome. To illustrate this approach, we conduct two instantiating controlled experiments to evaluate the impact of program execution time and seed-corpus origin on the code coverage of various fuzzers. In these studies, we find that both variables can have a statistically significant impact on the relative effectiveness of fuzzers. Furthermore, we see that these differences are large enough to result in changes in the final ranking of fuzzers in practice. For example, we observe that SymSan [15, 16] is significantly worse at covering new control flow edges than AFL [62] or Eclipser [18] on a particular program. But, it is significantly *better* than either tool when that program's execution speed is *only 100ms slower*, with all other variables held constant. We also see that initializing the starting corpus from prior fuzzing runs of different fuzzers [1, 62] can result in different benchmarking outcomes; changing the probability AFL will achieve higher coverage than LibFuzzer from $p = 0.85$ to $p = 0.15$, on the lcms benchmark program, for example. However, it may not always be practical, possible, or realistic to manipulate exactly one benchmark property at a time. For instance, changing only the coverage of the initial seed corpus while keeping the number of seeds may not be possible; inducing an artificial, random slowdown during the execution of a program may not be realistic. Hence, we propose a second methodology.

Randomization. Our second methodology is to randomize the benchmark configuration and apply non-parametric *multiple linear regression*. Here we suggest to vary all properties simultaneously and measure the impact of individual properties using the coefficients from a multiple linear regression. By including the choice of fuzzer as an interaction term, we can quantify the relative impact a property has on a particular fuzzer and compare it to the effect of that property on other fuzzers. Suppose on the current benchmark LibFuzzer beats AFL in code coverage achieved after 24 hours. Our model can determine the degree to which a benchmark property must change (e.g., by choosing larger programs), such that AFL improves to beat LibFuzzer. Using this methodology, benchmark maintainers and fuzzer developers will understand **when** some fuzzers perform better or worse in certain contexts and can account for confounding variables in their evaluations. We illustrate this methodology using a modification of the FuzzBench evaluation platform, choosing eight benchmark properties that we reasonably believed to impact the fuzzer ranking for our instantiating experiment.

Contributions. Concretely, this paper makes the following contributions:

(1) We argue that the outcome of a benchmarking procedure depends on the specific properties of the benchmark. Hence, we recommend to augment the outcome with a counterfactual analysis, specifying the conditions under which the outcome would change, to increase the soundness and utility of the evaluation.

(2) We propose a framework for quantifying the impact of benchmark properties on the outcome using control if practical or randomization otherwise. Specifically, we present a novel application of non-parametric regression analysis to keep the number of required experiments comparable to the traditional evaluation methodology.

(3) We instantiate this methodology in three experiments, demonstrating its application in a practical setting. As a byproduct of this illustration, we identify several novel variables which can have significant impacts on benchmarking outcomes such as the corpus origin, program execution speed, and initial corpus coverage.
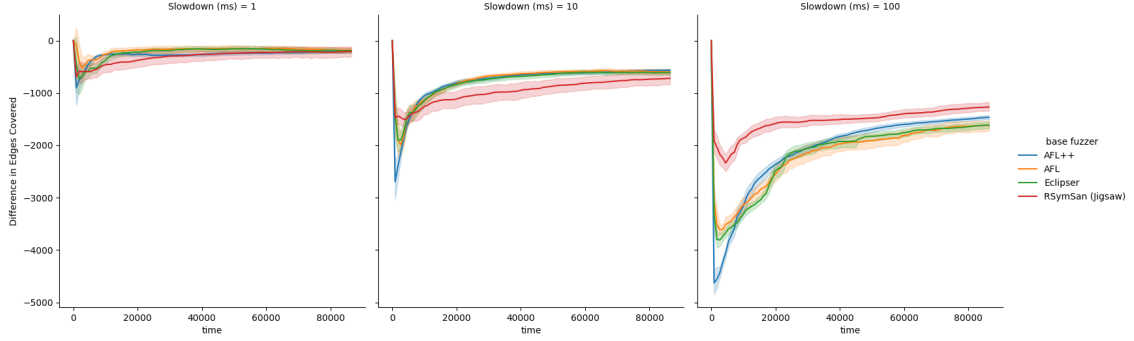
Fig. 1. Decrease in coverage relative to baseline at varying slowdowns of the target program

(4) We modify a leading benchmarking framework [49] to facilitate randomized experiments of arbitrary corpus properties, which we release publicly on acceptance.

In our instantiating experiments, we showcase our approach by answering four illustrative research questions. More specifically, using two *controlled* experiments, we investigate the following questions:

IRQ1 What is the impact of program execution speed on fuzzer effectiveness?
IRQ2 What is the impact of seed corpus origin on fuzzer effectiveness?

Next, we utilize *randomization* through the application of our non-parametric regression analysis, answering the following questions:

IRQ3 How is fuzzer ranking affected by varying a combination of properties?
IRQ4 Does seed initial corpus coverage affect the relative ranking of fuzzers?

2 CONTROL: MEASURING THE DEPENDENCE OF THE BENCHMARK OUTCOME ON ONE PROPERTY

Controlled experiments are the gold-standard for empirical research. By holding all other variables constant, researchers can estimate the impact of an explanatory variable by recording its correlation with a response variable of interest. In the context of fuzzing, a typical benchmarking setup is an example of a controlled experiment. All other variables –target programs, seed corpus, hardware etc.– are controlled while the explanatory variable, the fuzzer itself, is changed. The effect of each fuzzer is then evaluated with respect to different response variables, such as an ability to find bugs or code coverage. Because other variables (*covariates*) are controlled, the effect observed at the response variable is explained only by the effect induced by changing the explanatory variable.

However, if covariates do have an effect on the response variable, it can be important to understand what those effects are, especially if they interact with the explanatory variable. Control only allows researchers to isolate the effect of an explanatory variable for a *single configuration* of all other covariates. For example, if we fix the seed corpus to be empty in a controlled experiment, we might find that choosing AFL over LibFuzzer (explanatory variable) correlates with attaining higher code coverage (response variable)[34]. Because we controlled other variables, and we know that changing the fuzzer itself should change coverage achieved. We can even claim this to be a *causal* relationship; choosing AFL *causes* an increase in code coverage when the corpus is empty on the programs evaluated. In the following section, we will use controlled experiments to understand two previously unstudied covariates in fuzzing.

2.1 IRQ 1: Program Execution Time

Motivation. Much emphasis has been made in research on making fuzzers fast [29, 50, 59]; however, as we shall see, program execution time typically dominates greybox fuzzing campaigns. Different programs will have different runtime characteristics, including execution time. Thus it is important to understand how changes in the dominant portion of the fuzzing loop might affect existing fuzzers.

2.1.1 Experimental Setup.

Fuzzers. We select AFL [62], AFL++ [23], Eclipser [18] and SymSan [15, 16] as our four fuzzers. We select these fuzzers because they represent both state-of-the-art greybox fuzzers and varying levels of whitebox (symbolic) approaches to fuzzing. We hypothesize that different fuzzers, especially those using fundamentally different approaches, will be affected differently by faster or slower executing programs.

Benchmark. We fix the benchmark to a single program and fuzzing harness −libxml_xml− randomly selected from those included in the Fuzzbench [49] evaluation framework for a controlled experiment. Similarly, we use the default seed corpus for all 20 of our 24 hour trials for each fuzzer.

Hardware. We use a Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00 GHz for all experiments.

Explanatory and Response Variables. To vary the program execution time, we modify the fuzz driver for each fuzzer to inject a delay of 100ms, 10ms or 1ms in each iteration of the fuzzing loop. As our response variable, we investigate the delays' impact on code coverage.

2.1.2 Results. Figure 1 shows the net loss in edge coverage for each fuzzer at varying slowdowns of the target program. We can see that at a slowdown of only 1ms per execution (left), the fuzzers are all similarly affected, losing roughly the same amount of coverage each when compared to no slowdown. At a 10ms slowdown per execution (middle), again, all fuzzers lose roughly the same amount of edge-coverage. At this execution speed, we can also see that the greatest loss in edge coverage from slowing down the program for all fuzzers is at the beginning of a campaign, where there is a very steep dropoff in coverage. This gap then narrows approximately logarithmically over time for all fuzzers. When the program execution is slowed down by 100ms (right) we finally see a noticeable separation between the edge-coverage loss of each fuzzer. During the early stages of fuzzing when the edge-coverage loss is the largest, SymSan is substantially less affected by the slowdown relative to other fuzzers. By the 24 hour mark, SymSan remains the least affected by the slowdown, but only by a small margin over AFL++. While we see differences in the relative effect of program execution time on each fuzzer, from Figure 1 it is not clear whether these differences will significantly affect evaluation outcomes.

Figure 2 (a) and (b) show the pair-wise Vargha-Delaney $\hat{A}_{12}$[7] effect size between fuzzers with no slowdown (a) and with a slowdown of 100ms (b). Entries marked in **bold** are statistically significant by the Mann-Whitney U test ($p < 0.05$), adjusted for multiplicity (c.f. Section 4). For example, looking at Figure 2 (a), the probability that a random AFL run outperforms a random SymSan run is $\hat{A}_{12} = 83\%$. Below the effect sizes, we list the rankings of each fuzzer by final edge-coverage. It is clear from the rankings that the relative advantage gained by SymSan over the other fuzzers on slow programs is enough to completely change benchmarking outcomes. In terms of effect size, SymSan goes from only having a 12% chance of beating Eclipser in a fuzzing run to a 90% chance on a slower program! Similarly, SymSan flips the odds against AFL for slower programs by roughly 68 percentage points. However, despite SymSan being

	AFL	AFL++	Eclps	SymS
AFL		**0.00**	0.43	0.83
AFL++	**1.00**		**1.00**	**1.00**
Eclps	0.57	**1.00**		**0.88**
SymS	**0.19**	**0.00**	**0.12**	

Ranking
1.   AFL++
2. / 3.   AFL
    Eclipser (Eclps)
4.   SymSan (SymS)

(a) **Outcome of normal fuzzing run.**

	AFL	AFL++	Eclps	SymS
AFL		**0.00**	0.48	**0.13**
AFL++	**1.00**		**1.00**	**0.94**
Eclps	0.52	**0.00**		**0.11**
SymS	**0.87**	**0.06**	**0.90**	

Ranking
1.   AFL++
2.   SymSan (SymS)
3. / 4.   AFL
    Eclipser (Eclps)

(b) **Outcome if program was 100ms slower.**

Fig. 2. Pair-wise Vargha-Delaney $\hat{A}_{12}$ effect size between edge-coverage of fuzzers. Bold values indicate significance at $p < 0.05$.

relatively less affected by the slowdown than AFL++, this difference was not enough to make up for the large margin in edge-coverage achieved between AFL++ and SymSan with no slowdown.

We suspect that SymSan is the least affected by the slowdown because it is a partially symbolic approach; SymSan spends considerable time during the campaign solving path constraints with an SMT solver [16] and not executing the program itself. While Eclipser also focuses on constraint solving, it only does so in a lightweight, approximate manner for linear and monotonic constraints without invoking a solver [18]. This result may encourage renewed interest in symbolic techniques, which have generally struggled in recent years to outperform their simpler, but highly optimized greybox fuzzing counterparts on public benchmarks [45]. By expanding common fuzzing benchmarks to include slower programs, we might see a resurgence of these symbolic fuzzers. However, further research is needed to determine if these results generalize beyond our controlled experiment.

> *Program execution speed can have a significant effect on fuzzer efficacy, in absolute and relative terms. The symbolic fuzzer SymSan scales better as program execution time increases when all other variables are held constant for the* `libxml_xml` *benchmark.*

2.2 IRQ 2: Corpus Origin

Motivation. It is not uncommon to see evaluations where the corpus generated in an initial campaign of one fuzzer bootstraps a larger evaluation (e.g. [30]). If using seeds generated by such a bootstrapping campaign can bias fuzzer evaluations, this is an important, unaccounted for threat to the external validity of these research studies. Thus we propose a controlled experiment to determine whether the initial corpora origin affects fuzzers in practice.

2.2.1 Experimental Setup.

Fuzzers. We select AFL [62], AFL++ [23], LibFuzzer [1] and Entropic [10] as representative, state-of-the-art, general purpose greybox fuzzers for our controlled experiment from two different lineages (AFL and LibFuzzer). Our hypothesis is that corpora generated by the same fuzzer, or another similar fuzzer, will disadvantage that fuzzer in an evaluation against distinct fuzzers.

Benchmark. We fix the benchmark to a single program and fuzzing harness (`lcms-2017-03-21`) randomly selected from those included in the Fuzzbench[49] evaluation framework for a controlled 24 hour experiment (20 trials).

Hardware. We use a Intel(R) Xeon(R) Gold 6258R @ 2.70 GHz for all experiments.

	AFL	AFL++	EnLF	LF
AFL		0.27	0.41	**0.85**
AFL++	**0.73**		0.67	**0.94**
EnLF	0.59	0.33		**0.94**
LF	**0.15**	**0.06**	**0.06**	

Ranking
1. AFL++
2. Entropic (EnLF)
3. AFL
4. LibFuzzer (LF)

(a) **Outcome if started on LibFuzzer-generated seeds.**

	AFL	AFL++	EnLF	LF
AFL		**0.11**	**0.00**	**0.00**
AFL++	**0.89**		0.40	0.60
EnLF	**1.00**	0.60		**0.77**
LF	**1.00**	0.40	**0.23**	

Ranking
1. Entropic (EnLF)
2. LibFuzzer (LF)
3. AFL++
4. AFL

(b) **Outcome if started on AFL-generated seeds.**

	AFL	AFL++	EnLF	LF
Effect Size $(\hat{A}_{12})$	**0.0429**	**0.0800**	**0.1464**	**0.1800**
Statist. Sign. (U)	2.3e-09	3.7e-07	1.9e-05	1.1e-04

(c) **Intra-fuzzer effectiveness difference if the evaluation was started with AFL-generated seed corpora instead of LibFuzzer-generated ones.**

Fig. 3. Vargha-Delaney $\hat{A}_{12}$ effect size of edge-coverage between fuzzers. Bold values indicate significance at $p < 0.05$.

Explanatory and Response Variables. To vary the seed corpus origin, we sampled the starting corpus for each trial from two pools of seeds, one generated by AFL and one generated by LibFuzzer. To create these seed pools, we conducted two pre-fuzzing runs of 24 hours each using AFL and LibFuzzer respectively.

2.2.2 Results. Figure 3 shows the outcome of this experiment. The grids in subfigures (a) and (b) shows the pair-wise effect size measured using Vargha-Delaney's $\hat{A}_{12}$, highlighted in bold for statistical significance according to the Mann-Whitney U test (p < 0.05), adjusted for multiplicity (c.f. Section 4). On this program, with initial corpora generated by LibFuzzer, AFL++ performed best and LibFuzzer worst. However, the evaluation outcome is very different if we run those same fuzzers on corpora generated by AFL. Figure 3.b shows the ranking *if AFL-generated seeds were provided* instead. Now Entropic performs best while AFL performs worst. The probability that an arbitrary AFL run outperforms an arbitrary LibFuzzer run is $\hat{A}_{12} < 1\%$. *The choice of seed origin has a substantial impact on the benchmark outcome.* Figure 3.c further demonstrates this impact. All fuzzers perform worse on AFL-generated seeds ($\hat{A}_{12} < 18\%$). Yet, the impact on AFL/AFL++ is greatest ($\hat{A}_{12} < 8\%$): AFL-based fuzzers perform worse using AFL-generated seeds.

In addition to showing that the corpus origin can have a significant impact on evaluation outcomes, we can also begin to see some trends with this controlled experiment. Both fuzzers from the LibFuzzer lineage (LibFuzzer and Entropic) improve in ranking when run on AFL-generated corpora. Similarly, the fuzzers from the AFL lineage (AFL++ and AFL) improve on LibFuzzer-generated corpora. In other words, using seed inputs generated by a fuzzer not only can negatively impact that fuzzer, but also may negatively impact *other similar fuzzer implementations*. Given that many fuzzers presented in research are modifications of existing tools [10, 11, 23], this means that any evaluation utilizing fuzzer-generated seeds could be biased by this behavior. One explanation for this result is that similar fuzzers tend to explore similar behaviors in the program under test. If most of those behaviors are already covered by the initial corpus, it leaves less room for the fuzzer to explore during the evaluation.

> *Corpus origin can have a significant influence on fuzzer effectiveness. On the* `lcms` *benchmark program, we observe that tools used to generate the initial corpora for an evaluation are negatively affected in that evaluation. Even distinct fuzzer implementations based on the same lineage as the the initial corpus generator can be significantly negatively affected.*

2.3 Challenges of Control

While preferable, our methodology of control is not always applicable. Our controlled experiments demonstrate that the result of the evaluation *can change* depending on the specific configuration of the benchmark. However, there are many issues that can arise when applying this methodology in practice. In particular, there are two core reasons why a second methodology may be needed instead of or in conjunction with controlled experiments.

Specificity. In order to keep other variables like program size and execution time constant, the fully controlled experiments in **IRQ1** and **IRQ2** were only run for a single benchmark program and configuration. Thus, their results cannot be safely generalized beyond this configuration without additional experiments. Does execution time affect fuzzers on programs other than libxml? We can test our hypothesis on other benchmark programs, but this necessarily means that our setup is no longer fully controlled; now *both* the execution speed and the program itself are variables that may affect the response. If we naively aggregate the results on these different programs, the resulting variance will include the effects of the programs themselves, reducing our ability to distinguish statistically significant effects for our variable of interest. We also gain no insights into what general program characteristics might impact our response variable. For example, programs which are already very slow might be less impacted by a further reduction in execution speed, but a single aggregate statistic over several programs cannot capture this relationship. The methodology we propose in the remainder of this paper can aggregate data from multiple programs and give a nuanced view of program characteristics with a unified approach.

Dependence and Realism. Unlike the choice of seed origin, many benchmark properties are difficult or impossible to manipulate independently. For example, it might not be possible to manipulate the size of the fuzzer seed inputs while holding the seeds' validity, the coverage of the corpus, or the execution time of the program constant. In answering **IRQ1**, we manipulated execution time without otherwise modifying the instructions executed or control flow and while holding all other properties constant. This allows us to make strong claims about execution speed specifically having different effects on different tools. However, in practice, execution speed is often (but not *always*) a product of the number of instructions and branches in a program. Additional instructions and branches translate into larger, more complex formulae passed to an SMT solver for symbolic approaches. Thus, a program that is slower *because* of additional complicated control flow could easily result in SymSan performing relatively worse than AFL – the exact opposite of the results from our controlled experiment in **IRQ1**! Yet, increasing the number of instructions or branches in a program while holding all other variables constant is, again, extremely challenging: What if the locations of the inserted code has an impact? What if the inserted code impacts the reachability of parts of the existing program? Which instructions or branch conditions should be inserted to make the setup "realistic"? Rather than attempt to construct a completely controlled experiment that may end up being highly artificial or impractical, we give a secondary methodology for these scenarios that allows us to assess the impact of variables *without* controlling their values. This also makes findings by prior work (e.g. [34]) actionable, in that the influence of known or suspected covariates can be effectively *subtracted* from experimental outcomes to obtain an unbiased view of fuzzer performance.

3 RANDOMIZATION: MEASURING THE DEPENDENCE ON MULTIPLE BENCHMARK PROPERTIES

When control is not practical or possible, our fuzzer evaluation methodology consists of a *non-parametric regression analysis* of benchmarking data, using *bootstrapped confidence intervals* to test for statistical significance of results. By constructing a regression model, one can account for the effects of multiple variables simultaneously with respect to a

performance metric. As our technique does not assume a particular error or data distribution, it is widely applicable in tool evaluations, even on non-linear data with strong outliers. Finally, because regression models form the basis for ANOVA and hypothesis testing, our approach can be used to make rigorous claims about the statistical significance of results, in addition for being useful for exploratory analysis. To enable our holistic methodology, we propose to randomize, rather than control, benchmark properties for each run of the evaluation and record the properties of interest. For example, controlling initial corpus by holding it static across repetitions, as in traditional evaluation, reduces variance. However, it excludes effects of the corpus on the benchmarking outcomes in doing so.

Randomization. Unlike in typical fuzzing evaluations, we recommend randomly varying properties of the benchmark for each repetition of the experiment. For example, we can accomplish this for *corpus properties* by sampling each starting corpus from a larger pool of seed inputs.[2] We also suggest using a *matched-pairs* experimental design, such that each fuzzer is run on each configuration (e.g. starting corpus) to allow for direct comparisons. Other benchmark configuration parameters, such as the benchmark programs themselves, compilers used and architecture can also be randomized or enumerated where practical.

Rank transformation. For holistic fuzzer evaluation, we recommend applying the rank transformation to our gathered data, replacing each data value with its relative rank in the overall dataset. For example, the trial with the lowest coverage on a given program would have a rank-transformed value of 1, the second lowest 2, etc.

There are several reasons for studying the *ordinal association* rather than the numeric association between benchmark property and performance measure. First, we do not need to assume that the distribution of the data itself is normal or that the relationship between our explanatory and response variables is linear. In fuzzing we often observe extreme outliers or exponential effects which provide undue influence on a measure of the strength of the association (i.e., correlation). Secondly, we do not require value domains to substantially overlap across programs. For instance, we observe that a low initial coverage for one program can be a high initial coverage for another. Hence, we conduct our analysis on the the rank-transformed value *within* a program for corpus properies or across all programs w.r.t. program properties. Finally, rank transformations are used in non-parametric methods when values vary widely in scale [19, 35, 37, 42, 48]—like in automatic software testing. For instance, the Friedman test [37] describes a similar regression analysis against the ranks.

Regression Analysis. To quantify the combined effect of benchmark properties and fuzzers on benchmarking outcome, we propose a *multiple linear regression* of rank-transformed data as our holistic benchmarking technique. A linear regression model represents a response variable, e.g. coverage or fuzzer ranking, as a linear combination of several explanatory variables. Each regression coefficient can be considered as a measure of effect size for its corresponding variable: it indicates the degree to which that explanatory variable contributes to the response variable.

In the case of fuzzer benchmarking, we can give a general formula for a generic set of fuzzers, benchmarks, and properties. Let $P = \{p_i\}_{i=1}^n$ be a set of n benchmark properties and F be a set of fuzzers. An example of $p \in P$ is program size. We first choose a *reference configuration* $C = \langle f, \{v_i\}_{i=1}^n \rangle$ by selecting a *reference fuzzer* $f \in F$ and *reference values* v_i for every property $p_i \in P$. A multiple linear regression finds the intercept α and the regression coefficients β_p, γ_f, and $\omega_{p,f}$, such that

---

[2]In this case, Böhme and Falk [9] observed that a linear increase in coverage requires an exponential increase in the number of inputs. Hence, the *average* corpus, if sampled uniformly, might still be saturated. Instead, we recommend to sample from an exponential distribution centered at a desired percentage of the saturated corpus.

$$R = \alpha + \left[\sum_{p_i \in P} \beta_i X_i \right] + \left[\sum_{f \in F} \gamma_f Y_f \right] + \left[\sum_{p_i \in P} \sum_{f \in F} \omega_{i,f} X_i Y_f \right] \tag{1}$$

where X_i is the rank of the i-th property relative to the reference level $v_i \in C$, and where $Y_f \in \{0, 1\}$ are indicator variables such that $Y_f = 1$ if $f \in F$ was used instead of the reference fuzzer $f' \in C$. By fitting the regression model on benchmarking data, one obtains estimated values for each of the coefficients in the model, and the thus effect size of each variable. For instance, if $\beta_i > 0$, we say that an increase in X_i by one unit gives an increase in fuzzer ranking by β_i. Similarly, the coefficient ω_{pf} of an *interaction term* describes the additional contribution of the benchmark property X_p if the fuzzer f was used instead of the reference fuzzer (LibFuzzer).

Need for multiple regression. We recommend multiple-regression because it accomplishes our two primary goals: (1) it provides a way to conduct hypothesis testing for statistical significance and a measure of effect size and (2) can account for multiple variables in its analysis.

Regression analysis encompasses a wealth of techniques, such as a standard t-test or bootstrapping, which one can use for hypothesis testing. Similarly, the regression coefficients themselves represent a measure of relative effect size for their corresponding variables. Multiple regression also allows one to account for the effects of a variable as if all other variables were held constant, even if this fine-grained control is not feasible in practice. This last point is crucial in software evaluations where various aspects of the inputs and the programs cannot be manipulated in isolation from each other.

Additionally, regression models and ANOVA are a well-understood and flexible group of techniques which are standard in many fields for statistical analysis. While *hypothesis testing* is vital for researchers to establish empirical differences between fuzzers, *exploratory analysis* may be of more interest to practitioners. Even without an *a priori* hypothesis, a regression model and its corresponding coefficients can be used to approximately gauge the effects of many variables using the same experimental setup and analysis.

Bootstrapped confidence intervals. As noted by Arcuri and Briand [7], the assumption that errors are distributed normally is often violated in software engineering contexts because the data itself is not normally distributed. We address this concern by using *bootstrapping* as a method for computing non-parametric confidence intervals. By repeatedly regressing against sub-sampled data from our initial sample, we can obtain confidence intervals computationally, rather than analytically (avoiding the normality assumption).

Assumptions. Linear regression models make several assumptions about the underlying data, such as linearity, homoscedacity and independence. We outline how to check these assumptions in Section 3.3, and we guard against violating them with two other core aspects of our methodology: the *rank transformation* and *bootstrapping*.

3.1 Randomization Instantiation

To showcase our randomized holistic evaluation methodology, we *instantiate* it on a set of reasonable benchmark and initial corpus properties known or suspected to impact fuzzers. In doing so, we answer the following illustrative research questions using the data from a leading fuzzer evaluation platform,

IRQ3 How is fuzzer ranking affected by varying a *combination* of properties? **(Exploratory Analysis)**
IRQ4 Does initial corpus coverage *significantly* affect the *relative* ranking of fuzzers? **(Hypothesis Testing)**

Program	Description
freetype2	Font Renderer
harfbuzz	Text Shaping
libjpeg-turbo	JPEG Image Codec
libpcap (fuzz_both)	Packet Capture
libxslt (xpath)	XML Transformation
libpng	PNG Image Codec

Program	Description
sqlite3 (ossfuzz)	Embedded Database
vorbis	Audio Encoding
woff2	Web Fonts
zlib (uncompress)	Decompression
mbedtls (dtlsclient)	Cryptographic Primitives

Fig. 4. Benchmark Programs

3.1.1 Experimental Setup.

Hypothesis (IRQ4). Before running an experiment to test for a statistically significance, it is important to first formulate testable hypotheses. In this case, we use a reframing of IRQ4 as our instantiating hypotheses **H0.1-3**, but any other variable(s) in our model can be used.

H0: Changes in Initial Coverage have the same impact wrt. final coverage ranking on LibFuzzer as it does on...

(1) AFL,
(2) AFL++, and
(3) Entropic.

We recommend hypothesis testing on a single or small number of predetermined variable(s) to avoid Type I error (Section 4). For IRQ3, as we are only using the model for exploratory analysis, no formal hypotheses are needed.

Benchmark. (Figure 4). We chose the FuzzBench [49] platform to apply our holistic evaluation methodology because it is a leading tool with widespread usage among researchers and practitioners,[3] and there is large industry support from its corporate sponsor. Fuzzbench uses *branch coverage* as its measure of fuzzer performance. However, in principle our methodology applies to any other benchmark framework or performance measure [32, 49, 52].

Fuzzers. We chose AFL [62], AFL++ [23], LibFuzzer [1], and Entropic [10] (extends LibFuzzer), representing the state-of-the-art in general-purpose grey-box fuzzing.

Programs. To maximize the relevance of our findings, we use all programs directly integrated into FuzzBench[4] (i) that were already available in Commit 8858be7, (ii) that could be compiled and run within the local setup, and (iii) that had an OSS-Fuzz corpus available in Fuzzbench. Our benchmark consists of 11 programs listed in Figure 4. For every program, all experiments were conducted as FuzzBench local experiments on an AWS c6i.metal instance.

Benchmark properties. For our experiments, we select a set of reasonable properties, to *instantiate* our holistic benchmarking framework. We hope that this illustrative example analysis can both provide direct insights on the properties we evaluate and exemplify how future experiments can be conducted with new properties. We do not claim that these properties are exhaustive; we only investigate a subset of interesting properties which have been identified by prior work. As *corpus properties*, we measure the number of seeds, the initial LLVM branch coverage before the fuzzing campaign, the mean execution time in nanoseconds, and the mean size of the seeds in bytes. As *program properties*, we measure the size of the program, the proportion of equality and inequalities in comparison instructions, and the proportion of shared library calls.

We chose seed size and execution time as covariates as the documentation for AFL claims that each have a strong impact on performance [27][5]. We also record the number of shared library calls because instrumentation is the

---

[3]More than 140,000 CPU-days of public experiments run in 2022.
[4]Rather than adding additional programs or harnesses through the OSS-Fuzz Integration. Programs already added to Fuzzbench are widely used and thus results on these subjects are more directly relevant to evaluators
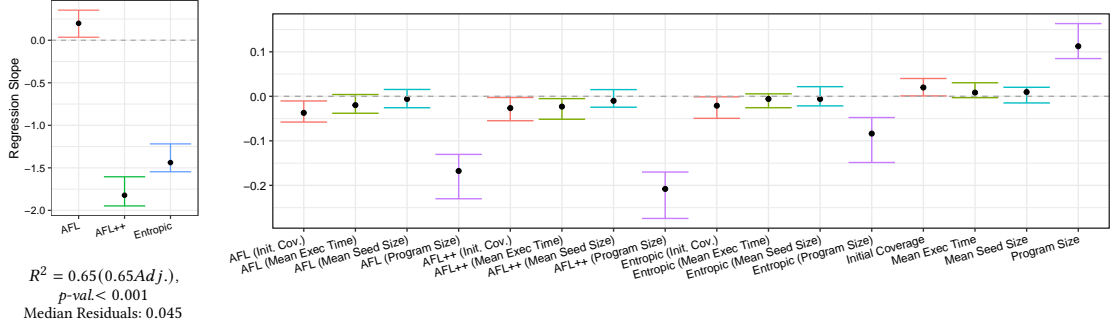
$R^2 = 0.65(0.65Adj.)$,
$p\text{-}val. < 0.001$
Median Residuals: 0.045

Fig. 5. Multiple Linear Regression with LibFuzzer as reference level (Fuzzer Ranking $\sim$ Fuzzer $\times$ Properties) [Eqn. 1].

defining characteristic of grey-box fuzzing techniques, shared libraries are not instrumented by default and are even recommended to leave uninstrumented [4]. We examine the impact of initial corpus size based off of observed effects in prior published work. For example, Klees et al. [41] show a difference between the empty and non-empty seed sets, but not for more granular changes in the size of the seed corpus. Additionally, some fuzzers like AFLFast have known issues with very large working corpora [63]. We measure initial coverage because it is well established that the performance characteristics change at different stages in a fuzzing campaign [9]. Similarly, we include program size as it impacts both the search space itself and how the fuzzer represents that space, which can lead to other performance issues such as those highlighted by CollAFL [25].

Instrumentation. To measure shared library calls and constraints in each program, we used `objdump` to extract the corresponding static instructions and their characteristics. We then used a binary instrumentation tool [21] to count these calls and constraints at runtime on the initial corpus. For program size, we report the size of the text segment in the program binary given by `objdump`.

Sampling configuration. For every {fuzzer, program}-pair, we conducted 24 trials of 24 hours. For each trial, we create the initial seed corpus by randomly sampling from a saturated seed corpus that was generated over many years by the OSS-Fuzz continuous fuzzing platform [2].[5] This highly saturated corpus approximates the "universe" of seed inputs. For the xth-trial ($x : 1 \leq x \leq 24$) of each program, all fuzzers start from the same initial corpus. Notably, we opt for a single trial per benchmark configuration. Running multiple trials per benchmark configuration would allow us to estimate the variance *not associated with the benchmark properties* themselves and compare to the variance when those properties are changed. This additional capability, however, comes at the cost of substantial additional computational resources (e.g. a factor of 20 for 20 trials per configuration). Using a single trial per configuration, we can still determine statistically significant results in terms of our response with respect to the overall variance. Given that fuzzer evaluations are already extremely costly, we believe that a single trial per configuration is likely to be preferred by most evaluators and thus choose this as our experimental setup.

3.2 Results

To analyze the results of our experiment, we instantiate the regression model introduced in Section 3 with our selected corpus properties and response variable (Eqn. 1). We define the following *reference configuration C*. As reference levels

---

[5]We use the FuzzBench option `-oss-fuzz-corpus` and minimize with `afl++-cmin`.

Benchmark Config 1	Benchmark Config 2	Benchmark Config 3	Benchmark Config 4	Official FB Config
↓ Low Initial Coverage	↓ Low Initial Coverage	− Median Initial Coverage	↑ High Initial Coverage	· Static seed set
↓ Small Programs	↑ Large Programs	− Median Sized Programs	↑ Large Programs	used by Fuzzbench
↓ Small and Fast Seeds	↓ Small and Fast Seeds	− Median Size and Speed Seeds	↑ Large and Slow Seeds	in all prior work
1. Entropic	1. AFL++	1. AFL++	1. AFL++	1. / 2. AFL++ / Entropic
LibFuzzer	Entropic	2. Entropic	2. Entropic	2. / 3. Entropic / AFL
3. AFL++	3. AFL	3. LibFuzzer	AFL	
4. AFL	LibFuzzer	4. AFL	4. LibFuzzer	4. LibFuzzer

Fig. 6. (left) Benchmarking outcomes at various levels of program and corpus properties (significant at bootstrapped 95% CI), (right) Benchmarking outcome from the Fuzzbench default corpora (significant at p < 0.05, Mann-Whitney U-test)

$\{v_i\}_{i=1}^{n} \in C$ we choose: for all corpus properties the lowest rank and for program size to the median rank. As reference fuzzer $f \in C$, we choose LibFuzzer.

Figure 5 shows the coefficients as point estimates and the non-parametric bootstrapped 95%-confidence intervals (CIs), adjusted for family-wise error rate as described in Section 4. If the CI does not include the origin (0), we can conclude that there is a statistically significant effect from the corresponding benchmark property relative to the reference levels at that level of confidence. The left side of Figure 5 shows the change in fuzzer rank if a different fuzzer is chosen and all other covariates (i.e., properties) are held constant at the reference level. The last four whiskers in Figure 5 (right side) show the change in fuzzer rank if one of the four considered properties increase by one rank and the fuzzer is held constant (LibFuzzer). The remaining whiskers show the *additional* change in fuzzer rank if both fuzzer and benchmark property are changed simultaneously — interaction terms in equation (1) presented earlier.

IRQ3: Exploratory Analysis of Properties in Combination

Looking at Figure 5, we see several substantial effects from our explanatory variables. On the left, we see that AFL appears to be outperformed by LibFuzzer, our reference fuzzer, at the reference levels for each of our covariates – its confidence interval is entirely above zero. Examining results on the right for corpus properties, we see that all three other fuzzers perform better with higher initial coverage corpora, as the confidence intervals are each below zero. Additionally, increasing mean execution time has a negative association with the performance of LibFuzzer relative to AFL++ (and possibly AFL as well). Finally, each other fuzzer appears to perform better than LibFuzzer as program size increases.

In light of these findings, we can gain some informal intuition about which fuzzers to use under which circumstances or formulate new, targeted hypotheses to focus on in future experiments. We might use AFL++ and Entropic and AFL over LibFuzzer for programs of size comparable to the medium and larger programs in Fuzzbench (median of 4917 instructions, max of 14452 instructions). However, on smaller programs with lower initial coverage and smaller or faster seeds, this gap in performance (coverage achieved) narrows significantly, and Entropic performs best, followed by AFL++ and LibFuzzer. However, for such conclusions to be empirically confirmed, we would need to run a new experiment to test each hypothesis, as will be exemplified in **IRQ4**.

Our holistic fuzzer evaluation methodology also allows us to put the negative impact of a benchmark property *in relation* to the impact of the choice of fuzzer at a reference level. Keeping all other covariates constant at their reference levels, a switch from LibFuzzer to AFL is roughly equivalent to increasing the program size from median size programs to the largest programs in our benchmark set. Comparing the coefficients for the fuzzer/corpus-property interactions, AFL++'s seem to be of similar magnitude to other fuzzers, suggesting that it is not excessively overfit to a particular starting corpus relative to other fuzzers.

Figure 6 shows the rankings given by our models at varying values of our measured corpus and program properties.[6] These rankings can easily be obtained from our model by adjusting the reference levels and fuzzers of Figure 5. The rankings are notably different depending on the values of our measured properties. LibFuzzer ranges from being among the best fuzzers (Figure 6, far left) to the worst (Figure 6, right). Similarly, AFL++, which has been highly optimized against the static Fuzzbench corpus falls as low as third. Furthermore, these rankings are only slices at discrete points in the space of possible evaluation configurations. One of the strengths of our holistic model is that it can produce such rankings at any point in the space, along with confidence intervals to gauge the significance of these results.

In contrast, the traditional evaluation setup can only give rankings for precisely one point in the space of configurations. We show the Fuzzbench [49] rankings for the four fuzzers we tested on the right of Figure 6. We compute the fuzzer rankings from the average run of each fuzzer across all our benchmark programs in existing experimental data, using the Mann-Whitney U-Test for significance ($p < 0.05$) [3]. The Fuzzbench rankings only represent the relative performance of these fuzzers for one initial corpus per program – the default seeds provided by Fuzzbench. The choices of initial corpora utilized by Fuzzbench are not explained, and seem to be arbitrary. Several target programs such as `jsoncpp` and `libjpeg` start with only a single input, yet others like `sqlite3` start with more than one thousand inputs. Given that we observe corpus properties have a significant effect on evaluation outcomes, this heavy usage of a single configuration is likely to introduce bias into the results. Indeed, we can see from the far right of Figure 6, that the rankings output by Fuzzbench are not the same as the rankings for representative median values of the properties that we observed in our study (Figure 6, middle). One could say that the Fuzzbench results (and thus fuzzers tuned on these results) may be *overfitted* to the Fuzzbench default seed set. The potential for this bias reinforces the need for evaluation platforms to utilize a wide variety of sampled starting corpora and benchmark programs as in this paper, rather than arbitrarily choosing a single evaluation configuration.

> *Our holistic model shows potential effects from the **program size** and **initial coverage** of the seed corpus, as well as some effects from the execution time of the seeds in the starting corpus on some fuzzers. These effects indicate how the final ranking of a fuzzer would change holding all other variables constant. Large variances in these properties appear to be enough to change fuzzer rankings in practice.*

IRQ4: Hypothesis Testing

Instead of an exploratory analysis, we can leverage our holistic model to do hypothesis testing. In this case, we use **H0.1-3** as our instantiating hypotheses, however the impact of any property could be tested in this way. Before testing multiple individual hypotheses, we first run ANOVA for our model to see if *any* variable effects are significant. Here we see a P-value of far less than our chosen significance level of 0.05 (Figure 5, left) and so we conclude that at least one of the variables in our model contributes significantly to our response variable.

To run a post-hoc test for a hypothesis using our holistic model, we can simply see if the confidence interval for the coefficient of the corresponding variable overlaps with zero. In this case we are looking at the relative effect of the initial coverage of the seed corpus for each fuzzer against LibFuzzer, which is captured by the interaction terms of (fuzzer*seed size). Looking at Figure 5, we can see that *none* of the confidence intervals reach zero. Thus we can reject the null hypothesis for **H0.1-3** and claim that initial coverage has a significantly different effect on other fuzzers relative to LibFuzzer.

---

[6]Differences in rankings is determined by the bootstrapped 95% confidence intervals, adjusted for multiplicity (c.f. Section 4)

Model	Acc. (%)	R^2	Adj. R^2	DoF
Atomistic (static rankings)	49.6	0.484	0.485	(3, 790)
Holistic [Eqn. 1]	59.5	0.669	0.661	(19, 774)
Extended Holistic [Eqn. 2]	67.6	0.780	0.763	(59, 734)

Fig. 7. Prediction accuracy and model statistics

To test other hypotheses between other fuzzers (e.g. AFL vs. Entropic), we could simply replot the data such that e.g. AFL, rather than LibFuzzer is the reference fuzzer.

> *The bootstrapped confidence intervals from our holistic regression model can be used to test individual hypotheses for significance. In our instantiating experiment, we would reject the null hypotheses and conclude that **initial coverage** has a statistically significant relative impact on AFL, AFL++, and Entropic relative to LibFuzzer.*

3.3 Assumptions and Model Validation

To ensure the correctness of our approach, we examine the fit and prediction accuracy of our regression model. We also check the general assumptions of regression analysis.

3.3.1 Model Validation. To evaluate the fit of our model we examined the prediction accuracy and mean-squared error relative to an atomistic model. The atomistic model predicts that a fuzzer's ranking on the training set will correspond to its ranking on the holdout set. This atomistic model represents the current state-of-the-practice in fuzzer evaluation – i.e. the average rank shown at the top of a Fuzzbench report.[7] We use a 75%-25% train-test split of the data from the previous section for evaluating prediction accuracy of our response variable, with 5-fold cross-validation on the training set.

Figure 7 shows the holdout-set prediction accuracy, goodness of fit (R^2), and degrees of freedom (DoF) for the traditional atomistic approach and our proposed holistic model. The traditional approach in the first row mirrors how users choose fuzzers today. This model chooses a ranking for unseen data based only on the overall rankings on prior data (the best fuzzer overall is always predicted to have ranking 1, the second best 2, etc.). For the holistic model in the second row, our multiple linear regression model (fitted on the seen benchmark configurations) predicts the benchmark outcome on an unseen configuration. We use the extended holistic model in the third row of Figure 7 to assess variance from program-specific effects, and will explain its precise nature in the coming section.

Given an unseen benchmark configuration, our holistic model can predict the correct ranking of a fuzzer roughly 60% of the time. In contrast, an atomistic model can only predict a fuzzer's ranking correctly less than half of the time on the unseen configurations. The holistic perspective thus gives users a substantially better chance of selecting a fuzzer that best fits their use case.

> *Our holistic methodology is nearly 20% more accurate in predicting the correct ranking of a fuzzer than a traditional evaluation (10 percentage points).*

Looking at the goodness-of-fit (R^2), we see the holistic model is substantially better than the traditional approach (+0.18 units). We can also remove the benchmark properties from our holistic model to assess their cumulative impact on evaluation outcomes in terms of R^2 value. Doing so creates a "fuzzer-only" regression model, which has an R^2 of

---

[7]https://www.fuzzbench.com/reports/sample/index.html

0.604 units. In other words, the properties explored in our investigation explain an additional 6.5 percentage points of the total variance in fuzzer rankings. However, 33% of the total variance in predicting the fuzzer ranking R remains unexplained by our model.

From the remaining variance, we hypothesize that there are significant program-specific effects not measured in our study, and thus not captured in our holistic model. By adjusting our model to take into account the benchmark program as a categorical variable, we can incorporate some of these effects in the aggregate. Let B be the set of programs. For each $b \in B$ and $f \in F$, let μ_b and $\rho_{b,f}$ be additional coefficients. Given reference level $C' = \langle f, b, \{p_i\}_{i=1}^n \rangle$, the fuzzer rank R is

$$R = \alpha + \left[\sum_{p \in P} \beta_p X_p\right] + \left[\sum_{f \in F} \gamma_f Y_f\right] + \left[\sum_{p \in P}\sum_{f \in F} \omega_{p,f} X_p Y_f\right] \qquad (2)$$
$$+ \left[\sum_{b \in B} \mu_b Z_b\right] + \left[\sum_{b \in B}\sum_{f \in F} \rho_{b,f} Z_b Y_f\right]$$

Here, Z_b is an indicator variable, s.t. $Z_b = 1$ if b was used instead of the reference program, and $Z_b = 0$ otherwise.

We trained and evaluated this extended holistic model on our split data-set. In terms of prediction accuracy, the extended model beats the traditional methodology by almost 20 percentage points, up to 67.6% on unseen data. Adding the program to the regression model explains an additional 6.4 percentage of variance over our initial holistic model, nearly 13% more variance than the "fuzzer-only" model.

These improvements indicate that there are still substantial *program-specific* effects that are not captured by the program properties recorded we recorded for this instantiating study. In other words, the coefficients depicted in Figure 5 give us a sense of the *net* effect of the corpus properties across programs, but individual effects vary substantially. For example, overall, Initial Corpus Coverage may have a significant effect on the relative performance of Libfuzzer vs. AFL as found in **IRQ4**. However, for some programs, this relationship may be somewhat weaker or stronger. Thus, it is crucial for benchmarking frameworks to maintain a large, representative sample of target programs to avoid systematic bias with respect to program and corpus properties.

This observation further motivates our proposed methodology in future work, as exploratory regression analysis (**IRQ3**) can identify pertinent benchmark properties and hypothesis testing (**IRQ4**) can confirm their effects. Because regression analysis is a sufficiently general technique, it is easy to incorporate additional benchmark properties—numeric, ordinal or categorical—as researchers identify them. We look forward to future holistic investigations, which examine additional properties to further explain these differences across programs.

3.3.2 Assumptions. To apply multiple linear regression, we require five assumptions to be met: (i) a linear association between explanatory and response variables [*linearity*], (ii) no high correlation among explanatory variables [*no multicolinearity*], (iii) a constant variance of residuals [*homoscedasticity*], (iv) normality of the residuals [*normality*] and (v) independence of observations [*independence*].

Linearity. For a multiple linear regression model to be accurate, the relationship between the explanatory and response variables must be correctly modeled with a linear formula. By using the rank transformation, we eliminate monotonic non-linearity, typically observed in fuzzing evaluations [9], from the model. We observe that all of our transformed predictors appear to be linear with respect to the fuzzer ranking in our supplementary materials (Section 7).
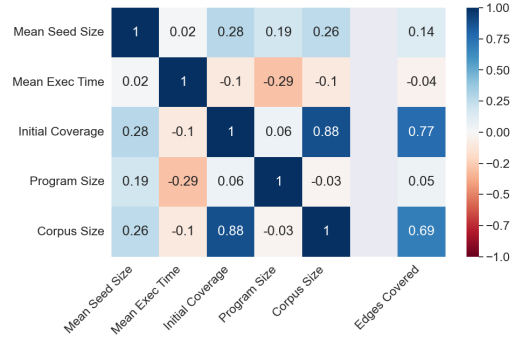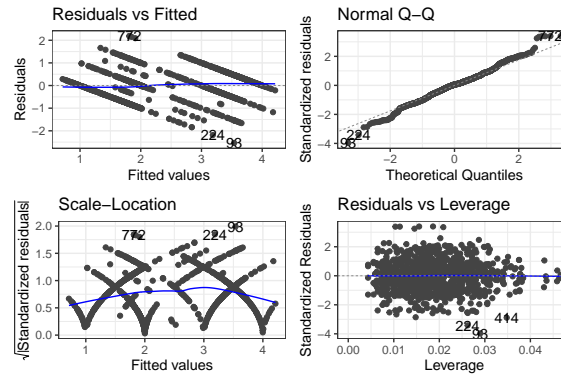
Fig. 8. Correlation matrix for Overall Impact Across Benchmarks and Fuzzers (Spearman's ρ).

Fig. 9. Model Diagnostics

Multicolinearity. If some explanatory variables are highly correlated, it becomes difficult to interpret the model as the joint effect can be arbitrarily spread among correlated predictors. In Figure 8, we can only see one association that is not weak or negligible, i.e., a very strong correlation between the size and coverage of the initial seed corpus. As a result, we exclude the corpus size from our model. For reasons of model parsimony, we also exclude the proportion of equalities, inequalities, and calls to shared library from the multiple linear regression model. These program properties have no discernible independent impact on individual or relative performance of our fuzzers.

Normality. To determine statistical significance of regression coefficients analytically, the residual errors of a linear regression model must be approximately normally distributed. Instead, we obtain confidence intervals *computationally* using a non-parametric bootstrap.

Homoscedacity. Linear regression models assume that the variance of the residuals is constant over the predicted values of the response variable. We can assess homoscedacity by examining the residual plot or the scale-location plot of the square root of the standardized residuals against the fitted values. In the latter, we should see a flat line if our data is homoscedastic. However, Figure 9 (left) shows that this might not be the case for our data. There seems to be greater variance around the middle ranks, likely due to ties in fuzzer performance (which we break by assigning the average rank). We therefore use the *wild bootstrap* to reduce the impact of heteroscedasticity on our confidence intervals [61] [46]. Intuitively, the wild bootstrap repeatedly resamples the residuals of a linear model fitted to the original and scales

them by a random variable to give additional data-points without making assumptions that the residuals are similarly distributed.

Independence. Regression models also assume that the individual samples of data are not dependent on each other. We use Fuzzbench for our testing setup, which should not introduce any dependence between trials. We validate the independence of the residuals of our data with the Durbin-Watts test [22], obtaining a p-value of 0.794.

3.3.3 Variable Selection and Model Parsimony. When constructing a statistical model for sample data, it is generally desirable to minimize model complexity when attempting to maximize the explanatory power of the model. Doing so makes the model more interpretable and also reduces the chances of overfitting to the sample data. Adding variables to a model will typically increase its effectiveness on the sample data, but this comes at the cost of making the model more complex. Balancing these two requirements is an inherent tradeoff in regression analysis.

Thus, in constructing our multiple regression model, we picked a *small* subset of properties that (1) we reasonably expected to have an impact on fuzzer effectiveness to demonstrate our approach (2) were easily measurable, (3) that were independent of each other to avoid issues with multicollinearity and overfitting and (4) that were well represented in our sample population of programs integrated with Fuzzbench. We identify several properties in Section 3.1.1 satisfying (1). To ensure (2), we omit other difficult-to-measure properties such as "seed validity" [34]. In pursuit of (3), we discarded several properties which were expected or that we found empirically to be highly correlated with others.[8] Specifically, we excluded the number of memory-unsafe accesses, cyclomatic complexity, and test suite size which are all known or expected to correlate strongly with program size [60]. We measured other program and corpus properties, including the quantiles of the execution times, quantiles of seed sizes in the initial corpus, as well as the number and types of constraints covered by the initial corpus [32]. However, we found these to be of negligible individual impact and thus excluded them from the final model. We refer the interested reader to the rich literature on variable selection [33]. Finally, we omit other variables –such as whether or not the program has an input "dictionary"– due to the small number of representatives in our sample population (4); without a significantly larger number of representatives, we cannot make any statistical claims about the impact of these variables.

What about important benchmark properties that are not explicitly modeled? These typically appear as additional unexplained variance in the fitted model (assuming the sample is representative). Like all statistical methods, our methodology can only account for variables present in the model and observed in the sample data. If the additional variables were included, the resulting model would have a greater explanatory power, but there would be a corresponding increase in model complexity and thus increased risk of overfitting our sample data. While there will almost always be *some* unexplained variance after fitting a model, researchers can make incremental improvements and reduce this unexplained variance by running new experiments and measuring new properties. When they do, they can use our methodology to determine whether these newly measured properties have a significant impact on evaluation outcomes independently of other variables.

4 THREATS TO VALIDITY

Like other empirical studies, our instantiating studies have several potential threats to validity which we have attempted to mitigate.

---

[8]We note that the discarded properties which are strongly correlated with others, such as cyclomatic complexity, may still have value in the model, but a much larger sample size would be necessary to separate out the smaller, individual effects of these variables.

Threats to internal validity. are aspects of our study that may introduce systemic bias. We minimize experimenter and confirmation bias in our experiment design by utilizing Fuzzbench, an existing benchmarking tool prepared by independent practitioners and researchers, for our experimental setup. We reduced the chance of selection bias in our choice of fuzzers by only choosing from state-of-the-art fuzzers with widespread adoption. Additionally, because we are not attempting to make claims about the performance of any particular fuzzer but rather *potential* covariates in benchmarking runs, impact of selection bias is reduced. Another possible source of bias was our use of AFL++'s corpus minimization tool to reduce our seed pool for each experiment, which could bias the initial minimized corpora. We mitigate this risk by only reducing with respect to edge-coverage, which we believe to be an uncontroversial criteria for corpus reduction across grey-box instrumentation tools. Alternative tools using LLVM coverage data were orders of magnitude slower, to the point that they are not practical for fuzzing practitioners. Additionally, prior work [34] has outlined the importance of corpus reduction for fuzzing effectiveness, and thus running experiments without minimized corpora would have threatened the external validity of our study. Finally, it is important to compare research claims to a baseline to demonstrate that these claims represent an improvement over existing alternatives. We are not aware of any alternative evaluation frameworks that incorporate and account for covariates in a non-parametric analysis. However, we do compare with the existing state-of-the-practice for fuzzer evaluations in Section 3.3.

Threats to external validity. could harm the generality of these results beyond the scope of our study. For our two controlled experiments, we only seek to demonstrate that certain specific properties *can* influence the outcomes of benchmarking runs. While we chose hypotheses for these experiments that we reasonably believe may generalize, we cannot say that our results for **IRQ1** and **IRQ2** will hold for other fuzzers or subject programs without additional experiments. Our holistic methodology in Section 3 describes how to account for these and other properties in practice on arbitrary other programs. For the third instantiating experiment in Section 3.2, we attempt to minimize this risk with respect to our benchmark programs by utilizing the Fuzzbench benchmark suite, which is broadly representative of open source software available to the fuzzing community [49]. There is a risk that by using an existing fuzzer benchmarking suite, our sample population *only* includes projects which have been previously subjected to heavy fuzzing; this may not be representative of general software programs, many of which have not been fuzzed. However, creating a new benchmark with programs that have never been fuzzed would greatly increase the cost of our study and risk introducing other biases in the benchmark (i.e. there may be reasons *why* these projects have not been fuzzed – e.g. they are very small or not widely used). As such, we opt to use a well-established benchmark for our initial study, with the hope that this methodology can be applied to other benchmarks and programs in future research. Additionally, we follow the recommendation of Böhme et al. [13], that ≥10 benchmark programs should be sufficient for coverage-based benchmarking, spending over 1000 CPU-hours gathering data for our analysis. We also make our data and experimental setup available for replication. We attempt to minimize external validity risk with respect to our fuzzers by choosing what we believe to be a representative set of state-of-the-art fuzzers. AFL and LibFuzzer are the baseline implementations for 10/11 state-of-the-art fuzzers supported in Fuzzbench's main experiment. AFL++ and Entropic are among the latest and highest performing variants of our two baseline implementations in recent benchmarking runs. Similarly, SymSan and Eclipser are both recently published tools at top research venues, with promising experimental results.

Threats to construct validity. concern whether or not the data collected in our study measure what we claim. The execution speed parameter assessed in **IRQ1** must be interpreted with caution. We fixed all other variables for this experiment, but in many real-world programs, execution time is correlated with other variables. Our results may

reasonably hold when other variables such as the number of instructions remain constant as execution time increases (e.g. if there is increased blocking on I/O resources).

In this study, we are examining the impact of various variables on "fuzzer effectiveness". The measure we used, code-coverage, has long been used as proxy for fuzzer performance, but there is a risk that our results do not translate to bug-finding effectiveness. However [13] found that coverage based benchmarking is correlated with bug finding ability. Additionally, because of the sparsity of bugs, using even the largest bug-based benchmarks available (around 200 bugs) as our metric for fuzzer effectiveness would threaten external validity of our study.

Threats to conclusion validity. may lead to misinterpretations of our data. We mitigate the risk by checking the assumptions of each technique used. In the case of our regression models, we also use the non-parametric wild bootstrap [61] to obtain confidence intervals, rather than analytical methods dependent on the distribution of the residuals.

Another potential threat to conclusion validity is our use of the rank transformation. We did so to reduce the impact of outliers, scale the data for each program such that it is comparable, and eliminate the impact of any monotonic non-linearity on our regression analysis. Indeed, without the rank transformation, our data appears to be non-linear with no obvious higher order pattern, in addition to having several strong outliers, and thus would not be amenable to regression analysis. While the rank transformation is a common statistical technique for non-parametric analysis [19, 35, 37, 42, 48], this transformation introduces a layer of indirection to our analysis; where we report correlations and trends in the ranked data, and not the underlying raw data.

Finally, testing multiple hypotheses can increase the probability of Type I error, known as the issue of multiplicity. While our holistic evaluation methodology can provide confidence intervals for an arbitrary number of effects, we advocate only formally testing a single or small number of hypothesis per experiment with our holistic methodology, as in 3.2. However, any time multiple tests are conducted, it is important to at least *consider* adjusting for multiplicity. Unfortunately, most adjustments typically decrease statistical power, often substantially so, and thus should not be applied blindly [51, 56]. As such, whether to conduct such a correction depends on the context and whether false negatives or false positives are more important to avoid. For academic research, usually false positives are less desirable than false negatives, so we use the Bonferroni correction for claims of significance in experiments for **IRQ 1** and **IRQ 2**. Note that because of our experimental setup, the loss in statistical power for these experiments is of minimal impact. For **IRQ 4**, we adjust the alphas for each regression coefficient's confidence interval using the Holm–Bonferroni [36] method, which is strictly more powerful than the standard Bonferroni correction.

5 RELATED WORK

In disciplines of computer science which often utilize empirical evaluations, the impact of the evaluation setup on its outcome has previously been noted, but—to the best of our knowledge—not systematically studied. We could not find other methodologies that *account* for or quantify the impact of the benchmark on the benchmarking outcome *during* the evaluation. For instance, Kudela recently found that 47 of 90 evolutionary algorithms exhibit a strong bias towards the center of the search space due to prominent benchmarks in that field having solutions at or near the center [43]. In machine learning, Japkowicz [39] warns that focus on performance measures in research might obscure important behaviors of the algorithms under consideration. In computer architecture, Panda et al. [53] study the characteristics of the SPEC2017 CPU benchmark suite primarily to identify a *subset* of benchmark programs and inputs that can approximate the results of the *entire* benchmark suite. These works all acknowledge an impact of the choice of benchmark on the outcome of the evaluation, but provide no guidance to account for these effects in practice.

Similarly, in automated software testing, Herrera et al. [34] found that the properties of the seed corpus had a substantial impact on the performance of a fuzzer. For example, they found that running fuzzers on the readelf program using i.e. an empty corpus, a single valid ELF file, and a large corpus of valid ELF files gave completely different results. This is an example of a *controlled* experiment where one variable, the corpus size, is changed while others are held constant. However, the control Herrera et al. exercise in their experiments is incomplete – the corpus size may be correlated or even confounding other important variables which cannot be held constant. Indeed, in the third instantiating study of our framework, we find that corpus size and the initial coverage of the corpus are highly correlated, both confirming and shedding additional light on their experiments. As a key takeaway from their work, Herrera et al. recommend that evaluators vary corpora to see how they impact the fuzzers being assessed. However, they not provide a methodology to investigate properties other than corpus size. Our work gives a concrete workflow that evaluators can use for arbitrary properties of the corpus or benchmark to make this recommendation actionable.

In terms of guidelines for the evaluation of automatic software testing tools (i.e., fuzzers), most recommendations are concerned with the sound statistical analysis of effect size and statistical significance. Arcuri and Briand [6] [7] provide an in-depth discussion of statistical, non-parametric measures that are particularly applicable in the context of automatic software testing. Klees et al. [41] later drew on these guidelines to identify specific issues with experimental design for fuzzer evaluations, such as lack of repetition, too few target programs, and lack of consideration for the impact of initial corpora. However, none of the previous guidelines provide clear steps for an evaluation methodology that incorporates these covariates, as presented in this paper.

Several works have emerged establishing benchmarks for fuzzing tools both generally, as in the case of Fuzzbench [49] and for specific classes of applications such ProFuzzBench [52]. UNIFUZZ [44] attempts to provide a comprehensive evaluation platform by studying the impact of fuzzer-specific factors like instrumentation methods and crash analysis tools. However it does not study the impact of benchmark properties. In fact, like other fuzzing benchmarks, UNIFUZZ uses a static seed set for each benchmark program. As a result it does not study the differences in fuzzer evaluation results that arise from the choice of benchmark properties. Magma [32] focuses on a suite of real-world bugs. Other projects have gathered data-sets of real-world bugs for the evaluation of various testing and debugging strategies [40] [31]. All of these works use methodology similar to Klees et al. [41] or Arcuri and Briand [7], thereby, neglecting the influence of benchmark properties on testing outcomes.

Synthetic benchmarks [20] [55] [57] [58] provide an alternative path for more holistic benchmarking. For example, Zhu et al. [63], propose creating artificial target programs with large numbers of features known or suspected to be problematic for fuzzers. They compare relative performance of two fuzzers on one such program in terms of artificial "bugs" found, observing a difference in behavior between AFL and AFLFast [11]. This approach, however, cannot account for different effects from the seed corpora for such artifical programs. Additionally, there remain doubts as to whether results from such synthetic benchmarks are representative of behavior in typical real-world programs [14] [26] [12]. Recent work by Lyu et al. [47] suggests program senstive energy allocation procedure to determine the power schedule of a fuzzer.

While our study focused on coverage as the primary metric for fuzzer performance, we expect program and corpus properties to have impacts beyond increased coverage. Another metric commonly used to evaluate testing tools is mutant injection [54]. The ground truth for testing effectiveness is bug-finding ability. Inozemtseva et al. [38] first investigated the correlation between code coverage and bugs found by Java unit testing suites. Böhme et al. [13] more recently found coverage to be a good objective function for fuzzers, given the sparsity of real-world bugs. We believe there are future research opportunities in assessing the impact of covariates with respect to these alternative metrics.

Ensemble Fuzzing. Ensemble fuzzing (a.k.a. collaborative fuzzing or fuzzer composition) is a related subfield of research. The goal of an ensemble is to select the best performing set of fuzzer(s) for a given workload. Our multiple-linear regression model could, in theory, be used to predict a set of fuzzers that perform best for a workload (indeed we use prediction accuracy to validate our model in Section 3.3), but this is not the primary purpose of our framework. We aim to provide developers and researchers with the tools to conduct nuanced evaluations of fuzzer performance; in essence to understand which variables contribute to a particular fuzzer performing best on a given benchmark, and account for the impacts of those variables. Additionally, existing ensemble fuzzers are more focused on *characteristics of the fuzzers* themselves rather than the properties of benchmark programs discussed in this work. For example, autofz [24] dynamically picks the fuzzer(s) making the best progress during an ongoing fuzz campaign, but cannot explain why those particular fuzzers are performing well (or why other fuzzers are not) on that particular workload. The EnFuzz [17] authors compose fuzzers looking for diversity in their broad characteristics, such as seed selection strategy or mutation strategy. Similarly, Cupid [30] empirically assesses which fuzzer pairs are most complementary in terms of code-coverage. One possible application of our methodology could be to shed further light into which program or corpus properties each individual fuzzer is leveraging in a successful ensemble.

6 PERSPECTIVE: BEYOND FUZZING

Benchmarking allows researchers and practitioners to make empirical claims about the properties of a newly proposed technology—fuzzers included. In this paper, we study how to assess the degree to which the outcome of an evaluation depends on the specific properties of the benchmark that is used for the evaluation. In particular, we propose using control and randomization to assess the effects of benchmark properties on the evaluation outcome. Our approach can be used to study arbitrary variable effects, both in isolation and in combination with each other. While we describe our approach in the context of fuzzing, this methodology can be applied to other evaluations in other domains.

Within our domain of expertise, we showcase three instantiating studies leveraging this methodology, finding several examples of statistically significant effects not accounted for in current evaluations. Fuzzing benchmarks should be diverse and randomized with respect to properties like program execution time, program size, seed origin, and initial coverage. Furthermore, we hope researchers will use holistic methods to identify additional properties of benchmarks which impact evaluation outcomes. Benchmarks such as Fuzzbench can incorporate our changes, randomizing initial corpora and other parameters to provide more robust outcomes. Finally, fuzzer developers can adopt our methodology to account for previously identified covariates in their evaluations.

7 DATA AVAILABILITY

We include our infrastructure, data, and analysis at:

<center>https://figshare.com/s/de961f6206f786997e87</center>

We will make these artifacts publicly available upon acceptance.

8 ACKNOWLEDGEMENTS

Manuscript submitted to ACM

REFERENCES

[1] [n. d.]. libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[2] 2017. *OSS-Fuzz: Google's Continuous Fuzzing Service for Open-Source Software*. USENIX Association, Vancouver, BC.

[3] 2021. Fuzzbench Paper Data. https://www.fuzzbench.com/reports/2021-04-23-paper/index.html. Accessed: 2023-01-23.

[4] 2022. AFLplusplus/readme.lto.md. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md Accessed: 2023-01-05.

[5] 2022. More about AFL. https://afl-1.readthedocs.io/en/latest/about_afl.html Accessed: 2023-01-05.

[6] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd international conference on software engineering (ICSE)*. IEEE, 1–10.

[7] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486

[8] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021).

[9] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 713–724.

[10] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as Markov chain. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[12] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 105–115. https://doi.org/10.1145/2610384.2628058

[13] 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)*.

[14] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. Association for Computing Machinery, 716–730.

[15] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. {SYMSAN}: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2531–2548.

[16] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 18–35.

[17] 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.

[18] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.

[19] William J Conover and Ronald L Iman. 1981. Rank transformations as a bridge between parametric and nonparametric statistics. *The American Statistician* 35, 3 (1981), 124–129.

[20] 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 (S&P)*. 110–121. https://doi.org/10.1109/SP.2016.15

[21] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–163.

[22] J. Durbin and G. S. Watson. 1971. Testing for Serial Correlation in Least Squares Regression. III. *Biometrika* 58, 1 (1971), 1–19.

[23] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. USENIX Association, USA, Article 10, 1 pages.

[24] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. 2023. autofz: automated fuzzer composition at runtime. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1901–1918.

[25] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[26] Sijia Geng, Yuekang Li, Yunlan Du, Jun Xu, Yang Liu, and Bing Mao. 2020. An Empirical Study on Benchmarks of Artificial Software Vulnerabilities. *CoRR* abs/2003.09561 (2020). arXiv:2003.09561 https://arxiv.org/abs/2003.09561

[27] Google. 2022. AFL Performance Tips. https://github.com/google/AFL/blob/master/docs/perf_tips.txt Accessed: 2023-01-05.

[28] Google. 2022. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz#trophies. Accessed: 2022-08-12.

[29] Rahul Gopinath and Andreas Zeller. 2019. Building fast fuzzers. *arXiv preprint arXiv:1911.07707* (2019).

[30] 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. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 360–372.

[31] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 90–101. https://doi.org/10.1109/ICST.2019.00019

[32] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[33] Georg Heinze, Christine Wallisch, and Daniela Dunkler. 2018. Variable selection–a review and recommendations for the practicing statistician. *Biometrical journal* 60, 3 (2018), 431–449.

[34] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery.

[35] T.P. Hettmansperger and J.W. McKean. 2010. *Robust Nonparametric Statistical Methods*. CRC Press. https://doi.org/10.1201/b10451

[36] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[37] Ronald L Iman and William J Conover. 1979. The use of the rank transform in regression. *Technometrics* 21, 4 (1979), 499–509.

[38] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*. 435–445.

[39] Nathalie Japkowicz. 2006. Why question machine learning evaluation methods. In *Proceedings of the 2006 AAAI (Workshop in Evaluation Methods for Machine Learning)*.

[40] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[41] 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.

[42] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621. https://doi.org/10.1080/01621459.1952.10483441

[43] Jakub Kudela. 2022. A critical problem in benchmarking and analysis of evolutionary computation methods. *Nature Machine Intelligence* (2022), 1–8.

[44] 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 *USENIX Security Symposium*. 2777–2794.

[45] Dongge Liu, Jonathan Metzman, Marcel Böhme, Oliver Chang, and Abhishek Arya. 2023. SBFT Tool Competition 2023–Fuzzing Track. *arXiv preprint arXiv:2304.10070* (2023).

[46] Regina Y Liu. 1988. Bootstrap procedures under some non-iid models. *The annals of statistics* 16, 4 (1988), 1696–1708.

[47] 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 *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[48] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

[49] 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.

[50] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

[51] Shinichi Nakagawa. 2004. A farewell to Bonferroni: the problems of low statistical power and publication bias. *Behavioral ecology* 15, 6 (2004), 1044–1045.

[52] Roberto Natella and Van-Thuan Pham. 2021. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[53] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. 2018. Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 271–282.

[54] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[55] Jibesh Patra and Michael Pradel. 2021. Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC-FSE)*. Association for Computing Machinery, New York, NY, USA.

[56] Thomas V Perneger. 1998. What's wrong with Bonferroni adjustments. *Bmj* 316, 7139 (1998), 1236–1238.

[57] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: Automated Bug Insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. Association for Computing Machinery, New York, NY, USA, 214–225. https://doi.org/10.1145/2991079.2991103

[58] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/3236024.3236084

[59] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. 2597–2614.

[60] Martin Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal* 3, 2 (1988), 30–36.

[61] Chien-Fu Jeff Wu. 1986. Jackknife, bootstrap and other resampling methods in regression analysis. *the Annals of Statistics* 14, 4 (1986), 1261–1295.

[62] Michal Zalewski. [n. d.]. American Fuzzy Lop (AFL). https://github.com/google/AFL.

[63] 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.