An Empirical Examination of Fuzzer Mutator Performance

James Kukucka
George Mason University
Fairfax, VA, USA
jkukucka@gmu.edu

Luís Pina
University of Illinois at Chicago
Chicago, IL, USA
luispina@uic.edu

Paul Ammann
George Mason University
Fairfax, VA, USA
pammann@gmu.edu

Jonathan Bell
Northeastern University
Boston, MA, USA
j.bell@northeastern.edu

Abstract

Over the past decade, hundreds of fuzzers have been published in top-tier security and software engineering conferences. Fuzzers are used to automatically test programs, ideally creating high-coverage input corpora and finding bugs. Modern “greybox” fuzzers evolve a corpus of inputs by applying *mutations* to inputs and then executing those new inputs while collecting coverage. New inputs that are “interesting” (e.g. reveal new coverage) are saved to the corpus. Given their non-deterministic nature, the impact of each design decision on the fuzzer’s performance can be difficult to predict. Some design decisions (e.g., " Should the fuzzer perform deterministic mutations of inputs? ") are exposed to end-users as configuration flags, but others (e.g., " What kinds of random mutations to apply to inputs?") are typically baked into the fuzzer code itself. This paper describes our over 12.5-CPU-year evaluation of the set of mutation operators employed by the popular AFL++ fuzzer, including the *havoc* phase, splicing, and RedQueen, exploring the impact of adjusting some of those unexposed configurations.

In this experience paper, we propose a methodology for determining different fuzzers’ behavioral diversity with respect to branch coverage and bug detection using rigorous statistical methods. Our key finding is that, across a range of targets, disabling certain mutation operators (some of which were previously “baked-in” to the fuzzer) resulted in inputs that cover different lines of code and reveal different bugs. A surprising result is disabling certain mutators leads to **more diverse** coverage and allows the fuzzer to find **more** bugs **faster**. We call for researchers to investigate seemingly simple design decisions in fuzzers more thoroughly and encourage fuzzer developers to expose more configuration parameters pertaining to these design decisions to end users.

CCS Concepts

• **Software and its engineering**;

Keywords

empirical studies, fuzzing evaluation, mutators

ACM Reference Format:
James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2024. An Empirical Examination of Fuzzer Mutator Performance. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680387

1 Introduction

Fuzz testing, or fuzzing, is an automatic testing technique that works by repeatedly mutating inputs and observing changes in coverage. It selects inputs that increase coverage and drops inputs that do not. Fuzzing is widely used by researchers and practitioners to find bugs and vulnerabilities. In fact, the most popular coverage-guided fuzzer, AFL [44], has found thousands of bugs in hundreds of different software projects, including critical technologies such as OpenSSL and the Linux Kernel. Naturally, as a result of this success, both industry and academia have actively contributed to novel fuzzing research over the past decade. There are currently hundreds of fuzzers published in top-tier security, software engineering, and hacker conferences [9, 15, 21, 30, 37–40, 44]. Fuzzer improvements are made incrementally, with most novel fuzzers being some sort of a derivative of another successful fuzzer.

One of fuzzing’s main strengths is its sheer speed, with hundreds or thousands of different inputs executed per second. Typically, the novelty in fuzzer development is a matter of guiding the search over the input space, proposing some novel mutator [34, 39], instrumentation [27, 35], or schedule optimization technique [11, 19, 47]. Mutators are important to guide the fuzzer toward inputs that reveal more coverage and, eventually, bugs. Fuzzers typically employ many different types of mutators, with different weights associated with each mutation. Furthermore, selecting which mutator to use is typically non-deterministic and follows a set of heuristics. However, rigorously evaluating *all* of the nuanced design decisions that go into constructing a fuzzer is hard. Which mutators should be used, and what probability weights should be assigned to each? How quickly should an input’s scheduling power decay? Each new fuzzing technique proposes a particular answer to these questions and empirically evaluates its approach as a combination of all the proposed changes that shows improvement over previous work. However, fuzzers are inherently probabilistic, and, as a result, a sound evaluation is extremely time and resource-intensive: best practices call for evaluations that take *decades* of CPU time [26, 31].

ISSTA ’24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680387

Understanding precisely how each individual feature interacts with each other is challenging, and as a result, many of these design decisions that "just work" are left unevaluated as long as the overall fuzzer performance appears to be "good enough." Furthermore, attempts to optimize these design decisions in one or more areas (*e.g.,* mutation) [30, 41] have been eclipsed through improvement in fuzzer scheduling or efficiency [32]. Therefore, it is important to understand the individual effects of such design decisions before investing time and resources into optimizing them.

A concrete example of one under-explored design decision is the *mutation* stage of the popular fuzzer AFL++ [19]. The mutation stage generates new inputs by making changes to interests that were previously found to be "interesting" and employs three high-level operations: *havoc*, *splicing*, and RedQueen [10]. The *havoc* stage repeatedly applies up to six different types of mutators to an existing input in an effort to generate a new, interesting input: bit flips, overwrites with random and "interesting" integers, block deletion, block duplication, and arithmetic operators. RedQueen leverages additional lightweight compile-time instrumentation to map input bytes to comparison operators throughout a program's runtime and mutate over those. Splicing merges two inputs together at a randomly selected point. AFL++ exposes RedQueen and splicing to the end-user in the form of environment variables, but all other havoc mutators are not individually controllable (they are all either on or off). Despite recent work on the overall impact of havoc as a whole [41], we still do not understand which mutators are effective in which circumstances or why. As a consequence, improving current havoc techniques is challenging due to the lack of foundational knowledge about the effectiveness of each mutation. To our knowledge, no fundamental study has been conducted on the effectiveness of these mutation operators, both singularly and in conjunction with each other, for finding bugs and achieving branch coverage.

A poorly performing mutation strategy can affect the fuzzer's output in different ways. Generating *uninteresting inputs* - *i.e.,* those that do not reveal new coverage - simply wastes the fuzzer's effort, making it take longer to find the same coverage and discover the same bugs. Generating *poor quality inputs* can guide the fuzzer towards low coverage improvements and away from finding bugs. Furthermore, discovering the relationship between input quality and fuzzer performance is very challenging. As a consequence, disabling mutations completely may lead to better fuzzer performance.

In this experience paper, we evaluate the relative efficacy of each mutator in isolation and compare their effectiveness against baseline AFL++. We first modify the baseline fuzzer AFL++ to enable/disable each havoc mutator independently and integrate this modified AFL++ into the Magma ground-truth fuzzing benchmark platform, where we expose the havoc mutations and splicing and RedQueen to the end-user. We conducted an extensive evaluation, using over 12.5 CPU-years of computation (Section 4). Surprisingly, our results suggest that disabling mutators actually results in better overall results based on the type of fuzzing target. To shed more light on the relationship between mutators and bugs found, we describe a deep dive into two known bugs that are reached and/or triggered more effectively by disabling mutators.

Based on our observations, we recommend a set of best practices for fuzzer developers in evaluating the impact of seemingly simple design decisions (Section 5), as well as how to best expose these features to end-users. We also recommend best practices for fuzzer users to enable or disable particular mutations based on the type of target they are interested in testing. We call on the fuzzing community to study and expose additional fuzzer configuration options, which have been hard-coded into the core fuzzing loop until now, thus enabling a broader range of studies on their effectiveness. We believe that our study paves the way for a large body of future work on the efficacy of mutators within fuzzer design, dynamic tuning of mutators throughout a fuzzing campaign, and improving on the practicality of mutator evaluation in terms of time and resources required.

Table 1: Categories of mutations that we toggle in our experiments. The "Lever" is the abbreviation that we use to refer to this mutator in our evaluation. Mutation operator names are defined in the AFL++ source code.

Lever	Mutation Type	Mutation Operators
N	HAVOC_USE_INSERT	EXTRA_OVERWRITE EXTRA_INSERT
D	HAVOC_USE_DELETE	DELETE
B	HAVOC_USE_BLOCKOPS	RAND8 CLONE OVERWRITE_COPY OVERWRITE_FIXED
A	HAVOC_USE_ARITHMETIC	ARITH8_ ARITH16_/+/BE ARITH132_/+/BE
I	HAVOC_USE_INTERESTING	INTERESTING8 INTERESTING16/BE INTERESTING32/BE
F	HAVOC_USE_BITFLIP	FLIP_BIT1

In summary, this paper makes the following contributions:

- We describe the design and implementation of an evaluation prototype on top of AFL++ that exposes previously "hard-coded" configurations;
- We describe a rigorous over-12.5-CPU year evaluation of AFL++ mutators: havoc, RedQueen, and splicing, and show that disabling mutators can actually result in statistically significantly *better* coverage and increased bugs found when compared to baseline AFL++ based on the type of target;
- We suggest best practices for fuzzer developers and researchers when determining the efficacy of new fuzzer mutators.

2 Background

Popular feedback-directed fuzzers like AFL/AFL++ [19, 44], libfuzzer [29], hongfuzz [22], and their derivative variants all contain a similar core fuzzing loop. The fuzzer begins with a corpus of seed inputs (provided by the user or generated randomly) and then enters a loop that selects an input from the current corpus, mutates it, and executes it. The fuzzer relies on feedback from execution (*e.g.,* coverage or other dynamic analysis) to decide on future input selection and mutation. Through this procedure, the fuzzer gradually evolves a corpus of inputs covering the application's behavior and, ideally, detects bugs. In this section, we provide a brief background on the three main ways that AFL++ mutates its inputs, as they are what we focus on in our evaluation.

2.1 Havoc

Havoc was first introduced in the seminal fuzzer AFL [44] and is a series of pseudorandomly stacked mutation operators. The core AFL algorithm can be separated into two mutation stages — deterministic and non-deterministic. The deterministic phase applies a series of mutations sequentially over candidate fuzzing inputs. These include walking sequential bitflips (*i.e.,* iterating over the file in a fixed interval, namely bits or bytes), walking sequential byteflips, simple arithmetic operations, and insertion of "interesting values" (*e.g.,* minimum/maximum integers, zeroes) [43]. Havoc is a non-deterministic stage that takes place after the deterministic stage fails to yield any new branch coverage utilizing a corpus of candidate fuzzing inputs. In AFL++, the deterministic stage is disabled by default, so havoc is the true workhorse of mutation in AFL++. The havoc algorithm consists of a pseudorandom stacking of the following mutations:

1. Single-bit flips
2. Attempts to set "interesting" bytes, words, or double words (considering both endians)
3. Addition or subtraction of small integers to bytes, words, or double words (considering both endians)
4. Completely random single-byte sets
5. Deletion of entire blocks of bytes
6. Block duplication via overwrite or insertion
7. Block memset to a constant value (zero or non-zero)

The number of times mutations are stacked is a pseudorandomly chosen power of two that is configured within the respective fuzzer's source code. In AFL++, the maximum number of stacked mutations within a single mutation iteration is 2^4, or 16. In early implementations of havoc, the distribution of mutation operators was uniformly random, but AFL++ introduced bias [19], as the developers explain: "Based on a fair amount of testing, the optimal execution path yields appear to be achieved when the probability of each operation is roughly the same; the number of stacked operations is chosen as a power-of-two between 1 and 64; and the block size for block operations is capped at around 1 kB" [43]. The bias used in AFL++'s havoc mutations' probabilities suggests that weighting them equally may actually *not* provide the optimal execution yield, which makes havoc mutations a prime target for evaluation. AFL (and AFL++) hard-codes parameters for the mutation stack size (how many mutators to apply at once), the size of each block that is mutated, and the probabilistic weights of each mutator. Note that this is not true when AFL++ is run in MOpt mode [30], but evaluating against MOpt mode is outside the scope of this evaluation, as we seek to understand the effect of individual mutations before attempting to optimize their use.

2.2 RedQueen

```
if( (uint64)input == (uint64)"DEADBEEF")
  panic();
```

Listing 1: Illustrative example of a comparison that may be tracked by RedQueen (adapted from [10])

The RedQueen strategy attempts to find interesting byte values by tracking comparisons as the target executes. For instance, consider the code in Listing 1. Using random mutations (*e.g.,* havoc described in Section 2.1) gives the fuzzer a minuscule chance of finding the sequence of bytes "DEADBEEF" that reveals the bug. Considering that the initial input string is "AAAAAAAABBBBBBBBAAAAAAAA", RedQueen first tracks the comparison "AAAAAAAA" == "DEADBEEF"; finds the compared value "DEADBEEF"; and produces mutations: "CEADBEEF", "EEADBEEF", and so on. It then places these mutations at the point in the input where "AAAAAAAA" occurs. Since there are multiple points in our input where that could happen, RedQueen uses a process called "colorization" to map the mutation to a specific 8-byte point in the input. Colorization replaces the interesting input pattern (*i.e.,* "AAAAAAAA") with different random strings (*e.g.,* "IEDHFGEYBBBBBBBBLAKSIBXJ"); and observes which random string makes it to the comparison being tracked. In this case, the next comparison is "IEDHFGEY" == "DEADBEEF"), which reveals the occurrence of "AAAAAAAA" that should be replaced. The RedQueen strategy is enabled by default in LLVM mode [8] of AFL++ and is referred to in AFL++ documentation as cmplog. More detailed information about the internals of RedQueen can be found in the original paper [10].

2.3 Splicing

The splicing mutation is a means of combining two parent inputs, with the goal of producing a more interesting result. For example, consider the two string inputs "ABCDEF" and "UVWXYZ". A possible splicing starts by choosing a midpoint at random, using the first input up to the midpoint, and then using the second input after the midpoint. In our example, selecting a midpoint value of 2 results in the spliced input "ABWXYZ". AFL++ performs splicing by taking two parent inputs from the input queue and combining them at a randomly selected midpoint [19]. Splicing is employed when havoc does not find any new interesting inputs. At this point, two inputs are spliced, and havoc is re-run on the spliced input.

3 Implementation

Modern fuzzers do not expose all possible parameters that control each stage of mutation. Our implementation identifies such hard-coded parameters, which we refer to as **levers**, and exposes them to the user with configurable on/off flags. In this work, we focus on three macro-level mutators — havoc, RedQueen, and splicing — and then delve deeper into individual mutation steps within the havoc mutator.

AFL++ groups havoc mutations into six specific categories. We decided to use each category as a lever and implemented code to toggle them on and off independently of each other. Table 1 describes each category and corresponding mutations. We configure the levers via a six-bit string defined as an environment variable. Each bit controls one mutation. For instance, 000000 indicates that havoc is totally turned off, and 111111 indicates that all mutations are active (*i.e.,* equivalent to the baseline AFL++ havoc mutator set). All bit strings in between follow the order of Table 1, *i.e.,* 100000 corresponds to category "N" being turned on and all others being turned off, etc. We chose to expose these levers as an environmental variable as it is a simple mechanism compatible with existing infrastructure for running fuzzing campaigns — *i.e.,* the Magma benchmarking framework [24].

We implement the levers themselves at compile time for two main reasons. First, it does not introduce any overhead at runtime. Second, and more importantly, it provides more compatibility with the Magma benchmarking framework, as it builds the fuzzer at the beginning of each run. The Magma benchmarking framework uses a build script for each fuzzer, which we modified by adding code to append a compiler preprocessor macro (using the gcc -D flag) to the AFL++ build command for each lever that was set to 1. Then, we modified the main fuzzing loop of AFL++, surrounding the various havoc stage mutation categories with #ifdef blocks. As a result, if a mutation category is disabled, it is not compiled into the current AFL++ variant that Magma is using.

We also used an environment variable to expose RedQueen as a lever — NO_CMPLOG. As before, Magma uses the lever in the build stage to specifically build AFL++ without RedQueen instrumentation support. As the AFL++ codebase already had support for disabling RedQueen at build time, our changes were limited to Magma.

The ability to turn splicing on or off is also already exposed by AFL++ as an environment variable — NO_SPLICING — which is on by default. Magma already supports controlling splicing via the environment variable and requires no extra changes.

4 Evaluation

Our empirical evaluation aims to answer the high-level question of whether the levers we selected impact the fuzzer's behavior. We evaluate performance using three overall metrics: number of branches covered, bugs reached, and bugs triggered. When comparing different fuzzer configurations, we consider not only the number of bugs reached or triggered by each fuzzer but also *which* bugs are reached or triggered and how often they are reproducible. As described in Section 3, we implement levers to enable/disable each havoc mutator, as well as RedQueen and splicing in isolation Rather than empirically evaluate every combination of levers, we restrict our evaluation to a macro-scale to evaluate the effects of havoc, splicing, and RedQueen by default and then perform a more fine-grained of only configurations with a single havoc mutator enabled at a time (with splicing and RedQueen remaining untouched). Examining the diversity of executions that result from different combinations of mutators may be exciting future work. Specifically, in this evaluation, we seek to answer the following research questions:

RQ1: Is there a significant difference in coverage between various AFL++ configuration combinations of normal Havoc, RedQueen and splicing?

RQ2: Is there a statistically significant difference in the bugs reached and triggered by various AFL++ configuration combinations of normal Havoc, RedQueen, and splicing?

RQ3: Is there a statistically significant difference in coverage between individual havoc mutators, both amongst themselves and compared to baseline normal havoc? Are there bugs that only individual havoc mutators can find?

4.1 Methodology

Evaluation Suite: We follow Klees et al.'s guidelines for best practices in fuzzer evaluation [26], choosing an evaluation suite with real-world target programs containing realistic, relevant bugs. We considered the FuzzBench [31] dataset, which is a curated set of open-source projects with fuzz harnesses that is effective for comparing fuzzers on code coverage but does not provide a ground-truth for bugs in the programs. Instead, we used the *Magma* ground-truth dataset [24], which consists of 21 benchmark programs collected from 8 target libraries. Magma contains 118 bugs, each of which was sourced from *real* bug reports that have been previously found and fixed by developers. Since each bug's location is known in advance, Magma's fuzzing infrastructure reports which bugs were triggered by each experiment and which bugs were reached (but not triggered). Reaching a unique bug simply means that the fuzzer was able to cover the code that contains a vulnerable condition, but it does not mean that the vulnerable condition was satisfied. *Triggering* a bug implies that the fuzzer was able to satisfy a vulnerable condition, thereby creating a defect-revealing input that results in a program crash or hang (*e.g.,* an integer must be 0 to cause a logic bug). Magma's fuzzing infrastructure also collects and reports branch coverage from each experiment. Table 2 shows a summary of the benchmark libraries and program versions and the file types they process.

Experiment Design and Execution: To characterize the effect of each lever and answer our research questions, we conduct fuzzing campaigns with each of the following configurations of AFL++ v3.15a:

1. **AFL++:** Unmodified AFL++. All levers enabled
2. **AFL++$_{\mathbf{HR}}$:** Havoc enabled, splicing disabled, RedQueen enabled
3. **AFL++$_{\mathbf{HS}}$:** Havoc and splicing enabled, RedQueen disabled
4. **AFL++$_{\mathbf{RS}}$:** Havoc disabled, splicing and RedQueen enabled
5. **AFL++$_{\mathbf{H}}$:** Havoc enabled, splicing and RedQueen disabled
6. **A,B,D,F,I,N:** Each of the havoc mutator levers enabled, one at a time (6 configurations), RedQueen and splicing enabled

Following best practices, each experiment consists of 20 repeated trials on each approach on each of the 21 benchmarks, each running for 24 hours [26]. Hence, in total, this experimental design required $(1 + 1 + 1 + 1 + 1 + 6) \times 20 \times 21 \times 24 = 110,880$ hours (12.65 years) of CPU time.[1] We execute our evaluation on a private cluster of Ubuntu 20.04 virtual machines, each with 4 vCPUs and 16GB RAM. We leverage the SLURM [42] utility on our computing cluster in conjunction with Magma to be able to efficiently run hundreds of configurations in parallel across the cluster of virtual machines while also being fault-tolerant. We run a single experiment at a time in each VM, ensuring that the overall hardware utilization and quality of service per VM remain constant. Each experiment runs

[1] These experiments were conducted at periods of low demand for resources. The cluster is powered entirely by renewable energy sources.

Table 2: Benchmark target libraries and programs, adopted from Magma [24].

Target	Programs	Version	File Type
libpng	`libpng_read_fuzzer`	1.6.38	PNG
libsndfile	`sndfile_fuzzer`	1.0.29	Audio files
libtiff	`tiff_read_rgba_fuzzer`, `tiffcp`	4.1.0	TIFF
libxml2	`libxml2_xml_read_memory_fuzzer`, `xmllint`	2.9.12	XML Documents
openssl	`asn1`, `asn1parse`, `client`, `server`, `bignum`, `x509`	3.0.0	Binary blobs
sqlite3	`sqlite3_fuzz`	3.32.0	SQL queries
php	`exif`, `json`, `parser`, `unserialize`	8.0.0-dev	Various
poppler	`pdf_fuzzer`, `pdfimages`, `pdftoppm`	0.88.0	PDF
lua	`lua`	5.4	Lua scripts

with an initial seed corpus supplied by the Magma project; experiments do not share corpora. Our replication package includes the container used to run these experiments, the raw output provided by each execution of Magma, and processed results [28]c.

Statistical Methodology: Given the probabilistic nature of fuzzing, it is important to apply a statistically rigorous methodology to determine if one fuzzer configuration truly behaves differently from another. As described above, we conducted 20 repeated trials of each fuzzer configuration on each target program. Each trial consists of multiple fuzzer executions (executed in parallel, one execution per VM), and we aggregate the branch and bug coverage of those multiple executions to define the coverage of that trial. To compute the total branch coverage of each configuration on a program target, we collect the average branch coverage across all trials. Without assuming normality, we apply a Mann-Whitney U-test to determine if the mean number of branches covered by each fuzzer configuration significantly differs from the baseline. In line with prior work, we show the total number of bugs detected by each approach. We also show the number of bugs that are *reliably* triggered by one fuzzer over another, computed by again using Fisher's exact test to determine if the number of trials in which that fuzzer configuration covered the bug is statistically significant from the baseline approach. We accept a result as significant only if $p < 0.01$. A recent survey on fuzzing evaluation by Shloegel et al.[36] calls for employing rigorous statistical methods in fuzzing evaluation but does not report on any fuzzing publications that do so for bugs. To the best of our knowledge, our evaluation is the first to propose this statistical method for fuzzers' bug-finding abilities.

4.2 RQ1: Is there a significant difference in coverage between various AFL++ configuration combinations of normal Havoc, RedQueen, and splicing?

Table 3 shows the total branch coverage for the configurations outlined in Section 4.1. Shaded cells represent statistically significant differences in total coverage when compared to the baseline AFL++. Cells shaded red indicate less coverage than AFL++, green indicates more. There are some noteworthy observations that can be made by quickly glancing at this table:

- The results suggest that including havoc has a statistically significant effect on the amount of code coverage that the fuzzer is able to obtain within a 24-hour period, as there is statistically significantly less coverage for the AFL++$_{\mathrm{RS}}$ configuration in 17/21 target programs.
- Turning off splicing suggests to either result in statistically insignificant differences in coverage, or, in the case of 7 target programs, statistically significantly *improved coverage* when compared to baseline AFL++. AFL++$_{\mathrm{H}}$ behaves similarly to AFL++$_{\mathrm{HR}}$ on most targets, resulting in statistically significantly more coverage than baseline AFL++ in 5/21 targets. However, not only does AFL++$_{\mathrm{HR}}$ statistically significantly improve coverage when compared to AFL++ in more targets, it is the best performing in terms of the targets that it does improve significantly (with one exception — sqlite3 — which AFL++$_{\mathrm{H}}$ performs slightly better on).
- For some targets (asn1parse and bignum within openssl), the coverage is the same for all configurations. We believe the observed result is likely due to the cryptographic nature of the targets and how the fuzzer may struggle to produce valid inputs. This is a signal to switch strategies or possibly create a new fuzzing harness.

We note that Table 3 does not provide any insight into the branches' diversity. We show those results when we examine the bug-finding capabilities of the various lever configurations in Section 4.3.

Table 3: Total branch coverage per program. Cells that are not shaded do not represent a statistically significant difference in total coverage (Table 4 show exact p-values). Cells shaded red indicate statistically significantly less coverage than AFL++, and blue indicates more. Values that are bolded indicate a large effect ($\hat{A}_{12} > 0.71$). All $\hat{A}_{12}$ values are available in our supplementary artifact).

		All Levers Enabled	Individual Havoc Levers						Two Levers Enabled			One Lever Enabled
Target	Program	AFL++	A	B	D	F	I	N	AFL++$_{RS}$	AFL++$_{HS}$	AFL++$_{HR}$	AFL++$_{H}$
libpng	libpng_read_fuzzer	1,513	1,250	**1,221**	**1,103**	**1,137**	**1,143**	**797**	1,272	**1,274**	1,524	**1,268**
libsndfile	sndfile_fuzzer	3,313	2,984	**2,979**	**2,878**	**2,920**	**3,017**	**2,174**	**2,785**	**3,046**	3,364	3,116
libtiff	tiff_read_rgba_fuzzer	3,446	3,223	**3,122**	**2,167**	**2,626**	**2,613**	**932**	**1,440**	3,497	3,572	3,463
libtiff	tiffcp	4,882	4,435	**4,549**	**3,117**	**4,127**	**4,264**	**1,528**	**2,820**	4,780	4,906	4,846
libxml2	libxml2_xml_read...	13,038	12,423	13,094	**12,642**	**12,360**	**12,285**	**9,931**	**11,080**	13,045	13,146	13,132
libxml2	xmllint	13,079	12,376	13,043	**12,626**	**12,284**	**12,234**	**10,250**	**11,860**	13,051	13,177	13,166
lua	lua	4,838	4,566	**4,727**	**4,765**	**4,590**	**4,460**	**4,070**	**4,650**	4,839	4,932	4,924
openssl	asn1	9,937	9,929	**9,522**	**9,849**	**9,894**	9,927	**9,750**	**9,763**	9,926	9,926	9,946
openssl	asn1parse	1,240	1,240	1,240	1,240	1,240	1,240	1,240	1,240	1,240	1,240	1,240
openssl	bignum	1,245	1,245	1,245	1,245	1,245	1,245	1,245	1,245	1,245	1,245	1,245
openssl	client	14,167	14,123	14,164	**14,080**	**14,126**	14,163	**14,006**	**14,141**	14,177	14,175	14,173
openssl	server	14,229	14,229	14,230	**14,209**	13,560	14,240	**14,204**	**14,220**	14,228	14,225	14,229
openssl	x509	7,910	7,908	**7,902**	**7,873**	7,912	7,911	**7,868**	**7,901**	7,910	7,915	7,911
php	exif	4,738	4,705	4,696	**4,653**	**4,660**	**4,630**	**4,199**	**4,465**	4,748	4,742	4,742
php	json	4,328	4,046	4,266	**4,141**	**4,079**	**4,143**	**3,918**	**4,074**	4,329	4,329	4,324
php	parser	18,347	18,167	18,402	**17,826**	**18,015**	**17,742**	**16,055**	**18,022**	18,334	18,407	18,392
php	unserialize	5,501	5,423	5,472	**5,090**	**5,366**	**5,268**	**4,700**	**5,212**	5,510	5,521	5,526
poppler	pdf_fuzzer	14,571	14,737	14,526	**13,614**	**14,403**	**14,298**	**10,933**	**12,532**	14,626	14,390	14,528
poppler	pdfimages	11,354	11,083	**11,105**	**10,056**	**10,916**	**10,736**	**7,663**	**9,870**	11,336	11,320	11,383
poppler	pdftoppm	12,559	12,661	**12,478**	**11,750**	**12,432**	**12,360**	**8,870**	**10,697**	12,636	12,725	12,718
sqlite3	sqlite3_fuzz	21,776	15,534	**19,838**	21,103	**16,714**	**11,080**	**5,029**	**1,849**	21,985	22,644	22,873

Table 4: P-values for the difference in total branch coverage per program as compared to the baseline AFL++ measure. These values support the comparisons displayed in Table 3.

		Individual Havoc Levers						Two Levers Enabled			One Lever Enabled
Target	Program	A	B	D	F	I	N	AFL++$_{RS}$	AFL++$_{HS}$	AFL++$_{HR}$	AFL++$_{H}$
libpng	libpng_read_fuzzer	0.0313	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.0132	0.0037	0.7861	0.0071
libsndfile	sndfile_fuzzer	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.2503	0.0514
libtiff	tiff_read_rgba_fuzzer	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.3369	0.0149	0.8604
libtiff	tiffcp	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.0106	0.5884	0.3167
libxml2	libxml2_xml_read...	<0.0001	2e-04	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.6456	<0.0001	<0.0001
libxml2	xmllint	<0.0001	0.0619	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.1134	2e-04	1e-04
lua	lua	<0.0001	<0.0001	0.006	<0.0001	<0.0001	<0.0001	<0.0001	0.8817	<0.0001	<0.0001
openssl	asn1	0.4093	3e-04	<0.0001	<0.0001	0.2912	<0.0001	<0.0001	0.2286	0.1593	0.2791
openssl	asn1parse	-	-	-	-	-	0.1624	-	-	-	-
openssl	bignum	-	-	-	-	-	-	-	-	-	-
openssl	client	<0.0001	0.946	<0.0001	<0.0001	0.3231	<0.0001	0.0039	0.49	0.5161	0.7453
openssl	server	0.5403	0.9565	<0.0001	0.0031	<0.0001	<0.0001	<0.0001	0.6714	0.1362	0.8265
openssl	x509	0.4239	<0.0001	<0.0001	0.1571	0.3486	<0.0001	<0.0001	0.9566	3e-04	0.4231
php	exif	<0.0001	0.0933	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.0383	0.3501	0.4647
php	json	<0.0001	0.3499	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.7043	0.5509	0.0831
php	parser	<0.0001	4e-04	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.4168	<0.0001	8e-04
php	unserialize	<0.0001	0.0699	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.3039	0.0764	0.0514
poppler	pdf_fuzzer	0.0498	0.0411	<0.0001	0.009	0.0013	<0.0001	<0.0001	0.3301	0.2674	0.5978
poppler	pdfimages	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	1	0.8711	0.4327
poppler	pdftoppm	0.1166	1e-04	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.3577	0.0066	0.0038
sqlite3	sqlite3_fuzz	<0.0001	<0.0001	0.0207	<0.0001	<0.0001	<0.0001	<0.0001	0.1404	<0.0001	<0.0001

Table 5: Reachability and triggerability of bugs per-configuration. Green means at least one bug is reached/triggered statistically more frequently, red means at least one bug is less, blue means at least one is more *and* at least one is less.

	Bugs Reached						Bugs Triggered					
target	AFL++	Mopt	AFL++$_{HS}$	AFL++$_{HR}$	AFL++$_{RS}$	AFL++$_{H}$	AFL++	Mopt	AFL++$_{HS}$	AFL++$_{HR}$	AFL++$_{RS}$	AFL++$_{H}$
libpng	6	6	6	6	6	6	3	2	2	4	1	3
libsndfile	8	8	8	8	6	8	7	7	7	7	4	7
libtiff	11	11	11	11	8	11	8	7	8	8	4	8
libxml2	9	9	8	9	8	9	6	6	4	6	2	7
lua	2	2	3	4	1	2	1	1	1	2	1	2
openssl	10	10	10	10	10	10	5	4	4	5	3	4
php	5	5	5	5	5	5	3	3	3	3	3	3
poppler	16	16	16	16	14	16	10	10	10	9	4	11
sqlite3	15	15	16	15	0	16	7	4	7	8	0	7
Total	82	82	83	84	58	83	50	44	46	52	22	52

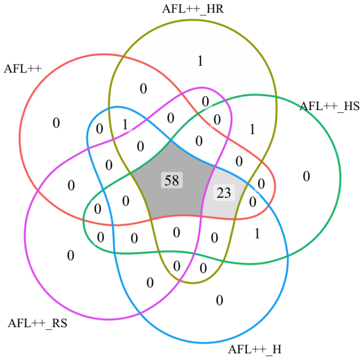


Figure 1: Bugs reached for baseline AFL++, AFL++$_{\text{HR}}$, AFL++$_{\text{HS}}$, AFL++$_{\text{RS}}$, and AFL++$_{\text{H}}$

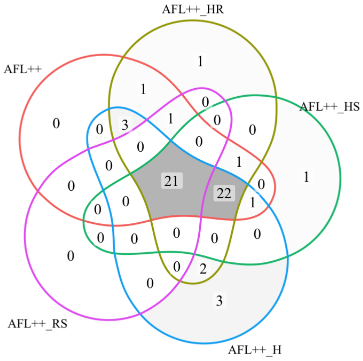


Figure 2: Bugs triggered for baseline AFL++, AFL++$_{\text{HR}}$, AFL++$_{\text{HS}}$, AFL++$_{\text{RS}}$, and AFL++$_{\text{H}}$

> **Answer to RQ1:** On a majority of targets, disabling havoc results in statistically significantly fewer branches covered than enabling havoc. Disabling splicing seemed to result in an equal amount of coverage or statistically more code coverage for all of the target programs. AFL++$_{\text{H}}$ followed a similar trend but did not increase coverage as much as AFL++$_{\text{HR}}$.

4.3 RQ2: Is there a statistically significant difference in the bugs reached and triggered by various AFL++ configuration combinations of normal Havoc, RedQueen, and splicing?

Figures 1 and 2 show Venn diagrams of bugs reached and triggered, respectively, for configurations 1–5 described in Section 4.1. We note that generating an input that triggers a bug does not necessarily imply increased branch coverage, but it does mean that the fuzzer produced an input with an unexpected value.

As Figure 1 shows, a majority of configurations were able to reach all of the bugs reachable amongst the configurations (81/85 or 95.29%), with a few outliers. The AFL++$_{\text{RS}}$ configuration was not able to reach 23 of those bugs and was not able to find any unique bugs, which indicates that turning havoc off severely hampers bug-finding capability.

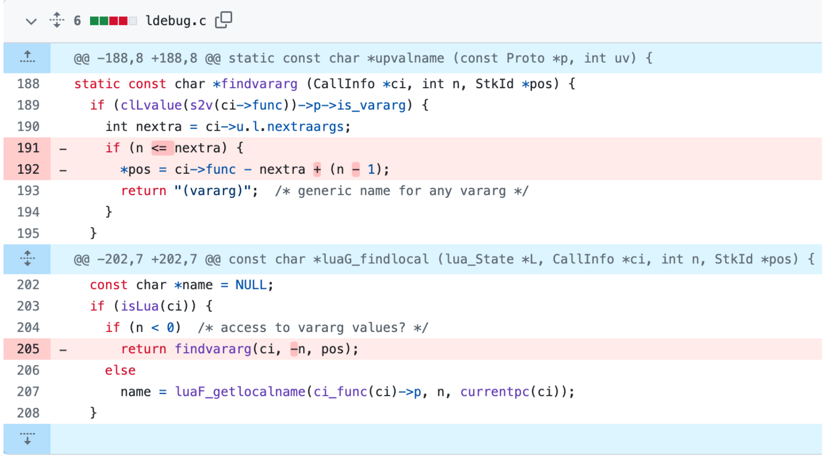
Interestingly, turning off splicing (AFL++$_{\text{HR}}$) reached one unique bug — LUA001 — which is a negation overflow that can lead to a segmentation fault in `getlocal` and `setlocal` functions [7], also reported as CVE-2020-24370. Figure 3 shows the actual location of this bug in the lua code where. Functions `setlocal` and `getLocal` call `luaG_findlocal` (Lines 202–208). In the code shown, the value `n` in the comparison on Line 204 is passed directly from the second argument of function `setlocal` to function `luaG_findlocal`. Therefore, any call to `setlocal` with a negative value as the second argument (*e.g.*, `setlocal(3,0x80000000)` or `setlocal(3,-1)`) results in a call to `findvararg` on Line 205 with the vulnerable condition. This is a very rare bug, as the fuzzer needs to mutate an input that calls `setlocal` (or `getlocal`) **and** passes a negative value to the second argument **and** does not violate the syntax of the function call. The odds of havoc finding a suitable input are minuscule. Splicing may actually exacerbate the problem because combining inputs may overwrite the call to `setlocal`. Further fuzzing of the spliced input will not cover the code path that leads to this bug. This type of bug is best suited for the RedQueen mutation, as it is able to map the location of the comparison of `n` to a specific point in an input and mutate there. Therefore, it makes sense that only the AFL++$_{\text{HR}}$ configuration was able to reach this bug. Examining Table 6, we can see that AFL++$_{\text{HR}}$ yielded almost double the amount of saved RedQueen inputs as baseline AFL++, indicating that it is useful at revealing new coverage and diversifying the fuzzing corpus. Additionally, the results suggest that RedQueen works in concert with havoc, as AFL++$_{\text{HR}}$ generated 17% more inputs than baseline AFL++. Furthermore, it appears that splicing may be an overall detriment to lua fuzzing, as 0 splicing-derived inputs are saved in the absence of havoc, and, when using AFL++$_{\text{H}}$, it generates the most saved inputs of any configuration. All configurations with splicing enabled generate fewer saved inputs than those without it (with one exception, AFL++$_{\text{RS}}$, which indicates that havoc is necessary to produce diverse inputs that increase code coverage).

Triggered bugs, shown in Figure 2, follow similar trends to reached bugs: a majority of triggered bugs (43/57 or 75.43%) were able to be triggered by every configuration. The AFL++$_{\text{HR}}$ configuration triggered one unique bug — LUA003, or ANH003 in the original Magma paper [24] — an API inconsistency bug in lua. The AFL++$_{\text{HS}}$ configuration triggered 1 unique bug – SQL007, or JCH220 in the original Magma paper [24] — a null pointer dereference within sqlite3. The AFL++$_{\text{H}}$ configuration triggered 3 unique bugs: (1) XML006 (or CVE-2017-9048 [2]) — a stack-based buffer overflow within libxml2, (2) LUA002 (or CVE-2020-24369 [5]) — a null pointer dereference within lua, and (3) PDF004 (or CVE-2019-10873 [4]) — a null pointer dereference within poppler.

A bug of particular interest is SQL007, triggered only by the AFL++$_{\text{HS}}$ configuration in our evaluation but not triggered at all by

Table 6: Total number saved, as well as number of inputs saved for each mutation type for the lua and sqlite3_fuzz target program.

Target	Program	Configuration	Total Inputs Saved	Havoc Inputs Saved	Splicing Inputs Saved	RedQueen Inputs Saved
lua	lua	AFL++	20363	14406	4776	1070
lua	lua	AFL++$_{RS}$	17524	0	0	17413
lua	lua	AFL++$_{HR}$	23949	21959	0	1878
lua	lua	AFL++$_{HS}$	21305	16314	4879	0
lua	lua	AFL++$_{H}$	25167	25054	0	0
sqlite3	sqlite3_fuzz	AFL++	49814	27297	12270	4912
sqlite3	sqlite3_fuzz	AFL++$_{RS}$	5951	0	0	616
sqlite3	sqlite3_fuzz	AFL++$_{HR}$	53628	40714	0	7579
sqlite3	sqlite3_fuzz	AFL++$_{HS}$	51317	33929	12048	0
sqlite3	sqlite3_fuzz	AFL++$_{H}$	55575	50235	0	0



```
@@ -188,8 +188,8 @@ static const char *upvalname (const Proto *p, int uv) {
  static const char *findvararg (CallInfo *ci, int n, StkId *pos) {
    if (clLvalue(s2v(ci->func))->p->is_vararg) {
      int nextra = ci->u.l.nextraargs;
-     if (n <= nextra) {
-       *pos = ci->func - nextra + (n - 1);
        return "(vararg)";  /* generic name for any vararg */
      }
    }
@@ -202,7 +202,7 @@ const char *luaG_findlocal (lua_State *L, CallInfo *ci, int n, StkId *pos) {
    const char *name = NULL;
    if (isLua(ci)) {
      if (n < 0)  /* access to vararg values? */
-       return findvararg(ci, -n, pos);
      else
        name = luaF_getlocalname(ci_func(ci)->p, n, currentpc(ci));
    }
```

Figure 3: Location of CVE-2020-24370 from a Github patch commit [6]. This bug was reached by the AFL++$_{HR}$ configuration and is only reached when passing a negative value to the `setlocal` function within lua.

AFL++ in the original Magma paper [24]. To exercise this bug, the input SQL statement must be a corrupted string having the form "CREATE TABLE ... AS SELECT ...". Valid inputs to sqlite3_fuzz are SQL statements, complicated strings without "magic values" used in comparison operators that would make good use of RedQueen. In this case, splicing and havoc are better suited to generate a string of that size and corrupt it, resulting in the null pointer dereference. This claim is backed up by the types of saved inputs for sqlite3_fuzz shown in Table 6, where for AFL++, RedQueen inputs only account for approximately 10% of total saved inputs. The AFL++$_{HS}$ total splicing inputs saved is comparable to that of the baseline AFL++, but there are over 6000 more saved havoc inputs than that of AFL++; which indicates that AFL++$_{HS}$ was able to make course-grained string mutations with splicing and havoc was able to mutate them to reveal new coverage. This claim is also supported by the fact that the AFL++$_{RS}$ configuration saved so few inputs. Splicing on its own was not enough to reveal much new coverage, and RedQueen clearly struggles to reveal new coverage on sqlite3_fuzz. Interestingly, despite finding this very unique bug, AFL++$_{HS}$ did not have statistically significantly different total branch coverage when compared to baseline AFL++, which corroborates the findings of Böhme et al. [12], where the fuzzers that trigger more bugs do not necessarily cover more branches and vice-versa.

Answer to RQ2: There are unique bugs that only certain configurations can reach, as certain mutations have a higher chance of finding those bugs. Time spent on other mutations does not help in finding such unique bugs.

4.4 RQ3: Is there a statistically significant difference in coverage between individual havoc mutators, both amongst themselves and compared to baseline normal havoc? Are there bugs that only individual havoc mutators can find?

Table 3 shows the coverage results for running each individual havoc mutator in isolation, with RedQueen and splicing still enabled. Across the vast majority of targets, using an individual havoc

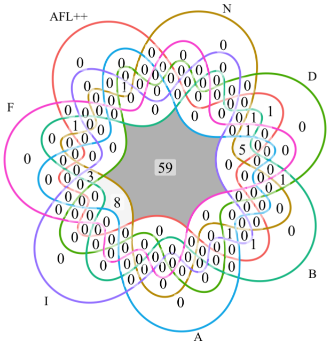


Figure 4: Bugs reached for individual havoc mutators vs. baseline AFL++

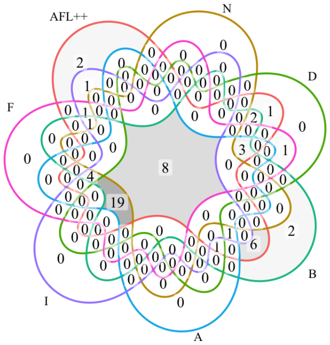


Figure 5: Bugs triggered for individual havoc mutators vs. baseline AFL++

mutator provided either statistically insignificant coverage or produced statistically significantly less coverage than the baseline AFL++. Furthermore, we performed statistical tests between the individual havoc mutators and determined that there were no statistically significant differences between them.

Of course, as we learned in Section 4.3, it is important to also consider bug-finding ability. Figures 4 and 5 show the bugs reached and triggered, respectively, for each individual havoc mutator level and baseline AFL++. According to Figure 4, no individual havoc mutator configuration reached any unique bugs. However, Figure 5 shows that baseline AFL++ was able to *trigger* two unique bugs, and the B configuration (block-operations-only in havoc) was able to trigger two unique bugs as well. The two unique bugs that the B configuration was able to trigger are: (1) PNG001 (or CVE-2018-13785 [3]) — an integer overflow and resultant divide-by-zero in libpng, and (2) PDF004 (or CVE-2019-10873 [4]) — a null pointer dereference within poppler. Both of these bugs were also triggered by configurations that leveraged havoc *other than AFL++* (*i.e.,* , AFL++$_{\text{HR}}$ and AFL++$_{\text{H}}$, but not AFL++$_{\text{HS}}$) in Section 4.3, which suggests that RedQueen and/or splicing hampered the baseline AFL++'s ability to trigger the bugs. The two bugs unique bugs that AFL++ was able to trigger are: (1) PNG006 — a dangling pointer bug in libpng [24], and (2) TIF008 (or CVE-2015-8784 [1]) — a heap buffer overflow within libtiff. The fact that AFL++ uniquely covered these two bugs suggests that interaction between individual havoc mutators is beneficial for satisfying vulnerable conditions within code. We discuss combinatorial relationships in Section 5.

> **Answer to RQ3:** There is no statistically significant difference in number of branches covered between individual havoc mutators. Across the vast majority of benchmark programs, reducing havoc to an individual mutator resulted in statistically significantly less coverage than baseline AFL++. Upon examining bugs, individual havoc mutators did not reach any unique bugs, but one configuration did trigger two, as did AFL++. This indicates that combinations of havoc mutators are beneficial to triggering bugs.

5 Discussion

In this section, we discuss the ramifications of our results and suggest best practices for developers and the fuzzing research community. We also discuss threats to validity and how we mitigate such threats.

5.1 Implications

For developers using fuzzers: Based on the results of this evaluation, we advise developers of fuzzers to start by examining the documentation of the fuzzer that they plan to use. As this study shows, incorporating more features (in this case, levers) counter-intuitively may not be the best approach for particular fuzzing targets. Furthermore, our results suggest that splicing may waste the fuzzer's effort and guide it away from interesting inputs, as it prevents the reaching and triggering of several potential vulnerabilities. When picking a new target to fuzz, we advise running a shorter preliminary phase with as many fuzzer configurations as possible for at least 24 hours to evaluate the efficacy of different configurable parameters in each fuzzer used. The results can then guide a much longer fuzzing campaign (which could be weeks or longer). Such a preliminary stage can shed light on which tools and configurations are more effective at achieving more code coverage and potentially finding more bugs for each particular target. Additionally, we advise developers to be skeptical of approaches that claim generality or an "optimal" approach, as "optimal" approaches may be only valid for a limited period of time, and alternative fuzzing strategies may prove to be more performant in the general case. This is evident in the example of HavocMAB vs. AFL++. HavocMAB sought to optimize the havoc stage of AFL by treating it as a multi-armed-bandit problem [41]. However, recent studies have found that AFL++'s latest features outperform HavocMAB in terms of coverage and bug-finding ability because it is based on AFL rather than AFL++ [32].

For the fuzzing research community: This empirical study characterizes a large body of future work that the fuzzing research community should be studying. Incremental fuzzing improvements are validated using a significant evaluation or study to determine a fuzzer's performance. However, new contributions devote little attention (if any) to important design decisions that end up being hard-coded in the final fuzzer. Such decisions should be evaluated in detail, taking each different target into account. We call on the fuzzing community at large to focus more on examining on these design decisions so that we may understand the current state of the art in all aspects of fuzzing — whether that be mutator selection, power scheduling, or some other aspect of the fuzzing algorithm. Additionally, when publishing new fuzzers, researchers should seek to document and expose as many design decisions as possible to the end-user in the form of levers — whether as compiler flags, environment variables, or configuration files. In this way, we can avoid the pitfalls of relying on community-accepted means of doing these things that are simply "baked-in." Once we understand seemingly simple design decisions, we can focus on broader research questions: How do we determine what are appropriate rewards to use for optimizing the selection of levers dynamically? What features of target programs should we be examining to determine how to alter fuzzing strategies dynamically? How do we automate this? How do we reuse information that we've learned to adapt to new versions of fuzzing targets or fuzzing targets that process similar inputs? These are all questions whose answers will improve the usability and adoptability of fuzzers in the future. In completing this study, we call on the fuzzing community at large to prioritize research towards this fundamental understanding so that we may answer these questions.

Additionally, we recognize that significant computing resources are required to perform a statistically significant evaluation, as shown in our evaluation which took over 12.5 years of CPU time. To evaluate all possible combinations of havoc mutators in our evaluation, we need to multiply that evaluation time approximately sixfold, as there are 64 combinations of individual havoc mutators. This is obviously quite costly for the everyday fuzzer developer, especially when real-world design decisions could be much more complicated. As such, we call on the community to conduct research on how to perform these evaluations more efficiently while maintaining statistical significance through heuristics or other means.

5.2 Threats to Validity

One concern of any experimental fuzzer evaluation is whether or not the results generalize to other combinations of levers and/or programs — i.e. will the findings here hold true if tested on new programs and with new levers? We posit that even if it does not, we have a strong result in that deactivating various mutations within the AFL++ fuzzing loop results in unique bugs being discovered and triggered. We expect that further exploration of more combinations of these same levers will continue to show interesting results. We sought to make our evaluation as simple as possible to illustrate this fact and avoid combinatoric explosion with regard to levers. Whether or not this generalizes to other target programs, we believe that the set of benchmarks shown in this evaluation represents a broad suite of targets, including targets that process highly and loosely structured inputs. We believe that the evaluation issue we are studying is a core issue that applies to any target, and we have already shown that results will vary across targets.

Another question that may arise from our evaluation is whether our results are statistically sound. To ensure that they are, we utilize best practices to ensure results are statistically significant with a high confidence interval (within 1%). When we present our data, we use appropriate statistical tests to compare means for differences. A rigorous statistical methodology for which unique branches and bugs are covered was something out of the scope of this evaluation. Instead, we use unique bugs reached and triggered as an illustrative example because unique reached bugs imply unique branches being covered. For future work studying unique branches in more detail, we advocate for rigorous, conservative statistical approaches, including using strategies such as the Bonferoni procedure to adjust p-values based on the number of comparisons made.

6 Related Work

Mutator Selection in Fuzzing: Prior work in mutation selection within a fuzzing run treats the problem as an optimization problem. MOPT [30] utilizes a customized Particle Swarm Optimization algorithm to find the optimal selection probability distribution of mutation operators with respect to fuzzing effectivenes. It achieves this by having multiple modules both perform fuzzing and continuously evaluating coverage instrumentation and program outputs to dynamically update the mutation scheduling probability distribution (*i.e.,* the "bias" that AFL++ places on each mutation operator). HavocMAB [41] seeks to achieve a similar goal by modeling the mutator selection process as a multi-armed bandit problem. It splits mutation operators into two categories - *chunk mutators* and *unit mutators*, and determines the category that would yield the maximum reward for each mutation cycle. After that, individual mutators are chosen uniformly randomly from the category. We chose not to compare AFL++ mutators to HavocMAB or MOPT because HavocMAB's evaluation determined that it outperformed MOPT [41], and recent studies have found that AFL++'s latest features outperform HavocMAB (even not when running MOpt mode), which is based on AFL rather than AFL++ [32]. The DARWIN fuzzer [25], published recently, uses an evolutionary algorithm to constantly modify the mutator selection algorithm in accordance with a reward score for the last mutation. This is similar to the way AFL++ itself selects inputs, as it assigns a reward score based on the previous input's code coverage.

Ensemble Methods: Swarm testing [23] was proposed in 2012 by Groce et al. as a way to improve the diversity of test cases during random testing, and is an inspiration for our evaluation. The idea behind swarm testing is not to create monolithic test cases that attempt to exercise all features of a program but rather to create a "swarm" of them that have some features omitted. Groce et al. showed that this leads to a better exploration of a program's state space. EnFuzz [16] and follow-on framework CollabFuzz [33] propose an ensemble method of fuzzing using multiple, diverse fuzzers and sharing an input corpus amongst all fuzzers. We did not conduct our evaluation using a shared seed pool because sharing a seed pool could hamper the progress made by these individual fuzzer variants, as it is possible that a scheduling algorithm could divert a variant's attention away from mutating on harder-to-cover branches in favor of progress made elsewhere.

Fuzzer Evaluation: One of the biggest problems facing the fuzzing community at large is that there currently are no universally agreed upon, standardized means of comparing the efficacy of different fuzzer implementations. The first large-scale investigation into the science of evaluating fuzzers was conducted in 2018 by Klees *et al.* [26]. In this work, the authors found problems in experimental evaluations conducted by 32 of the most recent, high-impact fuzzing papers. The paper reports a failure to do one or more of the following: establishing a compelling baseline fuzzer to compare against, establishing a sample of target programs for the purposes of the fuzzer, establishing a performance metric, establishing the set of configurations parameters (the seed files, length of the experiment, etc.), and accounting for the inherently stochastic nature of fuzzing. In an effort to mitigate this, Klees *et al.* were the first to suggest best practices for evaluating fuzzers [26], starting with a need to establish a uniform standard for benchmark programs and to conduct fuzzing evaluations for a minimum of 24 hours and with 20 or more runs for statistical significance.

Many published fuzzer evaluations use some combination of real-world programs, the LAVA-M dataset [18] or the Cyber Grand Challenge Qualifying Event (CQE) binaries [17]. The LAVA-M and CQE binaries contain a fixed number of synthetically introduced vulnerabilities and, therefore, provide some ground truth for evaluating fuzzers' ability to find bugs. However, the danger to using these datasets is the risk of overfitting — particularly due to the synthetic bugs. Additionally, the bugs within these datasets do not necessarily reflect the manifestation of real bugs in real programs. To rectify this, there have been advances in injecting bugs into programs that are representative of bugs found in real-world releases. The Magma [24] framework that we used in our evaluation patches current versions of benchmark programs with exemplar real-world vulnerabilities. Another more recent and sophisticated example is FixReverter [46] – a tool that automatically injects realistic bugs in a program by conducting both semantic and syntactical analysis to identify points where vulnerable code can be injected into a codebase. Using this methodology, the authors were able to create a benchmark suite of 10 programs with nearly 8000 bugs injected into them.

As indicated by our study, properly evaluating a fuzzer is expensive and time-consuming. Recent projects such as OSS-Fuzz [13], Magma [24] and FuzzBench [31] have significantly contributed to the state-of-the-art in terms of making fuzzer evaluation easier and more accessible to the open source community at large. OSS-Fuzz offers “fuzzing evaluation" as a service, with the primary goal of making fuzzing more accessible to open-source projects and to continuously fuzz these projects in the hopes of mitigating software vulnerabilities. FuzzBench [31] is built on OSS-Fuzz and is a state-of-the-art platform for evaluating fuzzers on a wide variety of targets at large scale. It is specifically built for fuzzing researchers to be able to compare their fuzzer with any number of variety of other fuzzers on OSS-Fuzz-supported targets. FuzzBench has indeed been adopted by the community as a state-of-the-art tool for fuzzer evaluation, and there have been a multitude of fuzzer evaluation studies that use FuzzBench [14, 20, 45, 46]. While FuzzBench allows for local experimentation, to run a full-scale experiment utilizing the framework, a fuzzer must be integrated into the FuzzBench repository, this makes it not necessarily the best tool for private developers testing a variety of experimental configurations. Magma [24] attempts to make fuzzer evaluation easier by providing a framework that evaluates fuzzers based on their ability to find a ground-truth set of synthetically introduced bugs. Magma's framework can be extended and run locally relatively easily, as we showed in this study. The most related FuzzBench evaluation to our work is the recent evaluation of AFL by Fioraldi et al. [20]. In this work, the authors took a more holistic approach to evaluating nine features of AFL, including power scheduling, scoring, trimming, and others. The goal of the work was to determine which features of AFL would be wise to include in all fuzzer improvements moving forward. The authors aimed to focus on not only traditional metrics of coverage and bug-finding but also usability and software reliability. Our evaluation differs in that it seeks to evaluate design decisions within the mutation stage of the most widely used fuzzer, AFL++. In doing so, we recommend general best practices for fuzzer mutation design, use, and evaluation moving forward.

7 Conclusion

In this experience paper, we present the first large-scale evaluation of individual mutation operators within the *havoc* stage of the AFL++ fuzzer, as well as the overall havoc, splicing, and RedQueen mutators. Our over 12 CPU-year evaluation showed that disabling certain mutators, particularly splicing, built into the AFL++ fuzzing loop allowed for more unique branches to be covered and unique bugs to be triggered. We also showed that while the number of branches between individual havoc mutators is not statistically significant, and running the mutators in isolation results in statistically significantly lower coverage when compared to the baseline AFL++, there does seem to be a benefit to combining them for bug finding. We presented best practices for developers using and writing fuzzers and call upon the research community to invest more time and resources into understanding seemingly simple design decisions within fuzzers and to conduct research towards improving the practicality of fuzzer evaluation. We believe our evaluation is a fundamental first step and is the beginning of a large body of work towards an understanding of fuzzer mutation strategies, and how those mutation strategies can be improved or dynamically modified to improve fuzzer performance, and therefore the quality of software as a whole.

8 Data Availability

Our artifact, consisting of the container used to run our experiments, the raw data that was output from the experiments, all source code of the modified AFL++ used in our evaluation, and our analysis scripts, is published along with this paper [28].

References

[1] [n.d.]. CVE - CVE-2015-8784 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8784.

[2] [n.d.]. CVE - CVE-2017-9048 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9048. [Accessed 11-04-2024].

[3] [n.d.]. CVE - CVE-2018-13785 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-13785.

[4] [n.d.]. CVE - CVE-2019-10873 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10873. [Accessed 11-04-2024].

[5] [n.d.]. CVE - CVE-2020-24369 — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-24369. [Accessed 11-04-2024].

[6] [n.d.]. Fixed bug: Negation overflow in getlocal/setlocal · lua/lua@a585eae — github.com. https://github.com/lua/lua/commit/a585eae6e7ada1ca9271607a4f48dfb17868ab7b. [Accessed 11-04-2024].

[7] [n.d.]. NVD - CVE-2020-24370 — nvd.nist.gov. https://nvd.nist.gov/vuln/detail/CVE-2020-24370. [Accessed 11-04-2024].

[8] [n.d.]. The LLVM Compiler Infrastructure Project — llvm.org. https://llvm.org. [Accessed 12-04-2024].

[9] 2023. Fuzzing Survey. https://fuzzing-survey.org/.

[10] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. [n.d.]. REDQUEEN: Fuzzing with Input-to-State Correspondence.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[12] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1621–1633. https://doi.org/10.1145/3510003.3510230

[13] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. *URL: https://github. com/google/ossfuzz* (2016).

[14] 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)*. USENIX Association, Boston, MA, 2531–2548. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju

[15] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725.

[16] 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)*. USENIX Association, Santa Clara, CA, 1967–1983. https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[17] DARPA. 2016. DARPA Cyber Grand Challenge Sample Challenges. https://github.com/CyberGrandChallenge/samples/.

[18] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

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

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

[21] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-Based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[22] Google. 2022. honggfuzz. https://honggfuzz.dev.

[23] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 78–88. https://doi.org/10.1145/2338965.2336763

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

[25] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2023. DARWIN: Survival of the Fittest Fuzzing Mutators. In *Proceedings 2023 Network and Distributed System Security Symposium*. Internet Society. https://doi.org/10.14722/ndss.2023.23159

[26] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[27] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 438–450. https://doi.org/10.1145/3510003.3510628

[28] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2024. *Artifact to accompany "An Empirical Examination of Fuzzer Mutator Performance" (ISSTA 2024 article)*. https://doi.org/10.5281/zenodo.12655683

[29] LLVM Project. 2019. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[31] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[32] Maria-Irina Nicolae, Max Eisele, and Andreas Zeller. 2023. Revisiting Neural Program Smoothing for Fuzzing *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 133–145. https://doi.org/10.1145/3611643.3616308

[33] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. CollabFuzz: A Framework for Collaborative Fuzzing *(EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3447852.3458720

[34] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*. https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf

[36] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. *arXiv preprint arXiv:2405.10220* (2024).

[37] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV*, Thomas Ball and Robert B. Jones (Eds.). 419–423.

[38] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[39] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing *(ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[40] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2011. Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution. *ACM Trans. Inf. Syst. Secur.* 14, 2, Article Article 15 (Sept. 2011), 28 pages. https://doi.org/10.1145/2019599.2019600

[41] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1634–1645. https://doi.org/10.1145/3510003.3510174

[42] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper 9*. Springer, 44–60.

[43] Michał Zalewski. 2014. Binary fuzzing strategies: What works, what doesn't. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html

[44] Michal Zalewski. 2019. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[45] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 53 (mar 2023), 21 pages. https://doi.org/10.1145/3580597

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

[47] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.ndss-symposium.org/ndss-paper/send-hardest-problems-my-way-probabilistic-path-prioritization-for-hybrid-fuzzing/

Received 2024-04-12; accepted 2024-07-03