Dissecting American Fuzzy Lop – A FuzzBench Evaluation

ANDREA FIORALDI, EURECOM, France
ALESSANDRO MANTOVANI, EURECOM, France
DOMINIK MAIER, Technische Universität Berlin, Germany
DAVIDE BALZAROTTI, EURECOM, France

AFL is one of the most used and extended fuzzer, adopted by industry and academic researchers alike. While the community agrees on AFL's effectiveness at discovering new vulnerabilities and at its outstanding usability, many of its internal design choices remain untested to date. Security practitioners often clone the project "as-is" and use it as a starting point to develop new techniques, usually taking everything under the hood for granted. Instead, we believe that a careful analysis of the different parameters could help modern fuzzers to improve their performance and explain how each choice can affect the outcome of security testing, either negatively or positively.

The goal of this paper is to provide a comprehensive understanding of the internal mechanisms of AFL by performing experiments and by comparing different metrics used to evaluate fuzzers. This can help to show the effectiveness of some techniques and to clarify which aspects are instead outdated. To perform our study we performed nine unique experiments that we carried out on the popular Fuzzbench platform. Each test focuses on a different aspect of AFL, ranging from its mutation approach to the feedback encoding scheme and its scheduling methodologies.

Our findings show that each design choice affects different factors of AFL. While some of these are positively correlated with the number of detected bugs or the coverage of the target application, other features are instead related to usability and reliability. Most important, we believe that the outcome of our experiments indicates which parts of AFL we should preserve in the design of modern fuzzers.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: fuzzing, afl, fuzzbench

1 INTRODUCTION

Recent research in software vulnerability discovery has identified *fuzzing*, or *fuzz testing*, as a key technology to efficiently detect bugs in different types of applications, including classical user-space programs [28, 31], OS kernels [40, 48, 49] and virtual machine hypervisors [39].

The high demand for more and more advanced fuzzers has resulted in a large proliferation of new prototype implementations. Some of these solutions have become well-known and largely adopted tools. Others have contributed to the research process, by studying new ideas that help fuzzers to uncover new vulnerabilities faster or with higher precision. Although every new tool comes with new features that distinguish it from existing fuzzers, a considerable amount of the functionalities is usually inherited from its "parent" project, which is often a well-established tool in the community.

Over the past five years, both industrial and academic research on fuzz testing has reached a consensus on a de-facto standard for fuzzing – the American Fuzzy Lop (AFL) [55] released in 2013 by Michał Zalewski. Two main aspects

Authors' addresses: Andrea Fioraldi, andrea.fioraldi@eurecom.fr, EURECOM, Campus SophiaTech, 450 Route des Chappes, Biot, France, 06410; Alessandro Mantovani, alessandro.mantovani@eurecom.fr, EURECOM, Campus SophiaTech, 450 Route des Chappes, Biot, France, 06410; Dominik Maier, dmaier@sect.tu-berlin.de, Technische Universität Berlin, Straße des 17. Juni 135, Berlin, Germany, 10623; Davide Balzarotti, davide.balzarotti@eurecom.fr, EURECOM, Campus SophiaTech, 450 Route des Chappes, Biot, France, 06410.

can explain AFL's success. On the one hand, its usability allows researchers to run the fuzzer out-of-the-box against several programs without any specific domain knowledge of the target itself. On the other hand, AFL excels at finding vulnerabilities fully automated, with low manual effort for security analysts. While these two factors are essential to explain the large success of this project, its development process passed through many phases of implementation and optimization. Often, new features are developed by multiple external contributors, with the inherent consequence that many design choices are not documented in a single and accessible resource.

This paper provides an accurate analysis of internal mechanisms, parameters, and algorithms, that determine the final behavior of the American Fuzzy Lop. In other words, we shed light on the design choices that have been implemented over the years and on their impact. In many cases, improvements came from contributions outside the academic ecosystem, thus lacking experiments and clear results to demonstrate why the author chose a specific technique over alternative options. As a result, today, everybody uses AFL without a complete understanding of its internals. However, we found that even minor modifications of the inner parameters affect the results of a fuzzing experiment, both positively and negatively.

More importantly, this lack of documentation prevents researchers from identifying, in a rigorous way, what the root causes behind the excellent performance of AFL are. We believe that this deep understanding is a fundamental step to guide future work in the field.

It is also important to understand that not all design choices are related to the effectiveness of the vulnerability discovery process. Some may instead improve other aspects of the fuzzing workflow, such as usability and reproducibility of results. In this paper, we also study whether these features are still beneficial in modern fuzzing campaigns or if they should now be considered outdated.

Our work's primary focus is on the algorithmic components that AFL embeds and that we can still find in other modern fuzzers, like its scheduler, the mutation engine, and the feedback mechanism. We exclude other specific engineering decisions, such as AFL's original solution to scale over multiple cores and machines. To verify the impact of each component, we performed a dedicated set of experiments in which we compare the vanilla AFL solution with a carefully-designed patched version of the project that replaces the feature under analysis. For instance, one of the mechanisms that captured our interest since the beginning was AFL use of *hitcounts* to encode the feedback in the coverage map. To study this aspect we patched AFL to include an alternative approach to measure the coverage, namely *plain edge coverage*.

Overall, we identify *nine* unique aspects that represent, to the best of our knowledge, the core design choices of modern fuzzers. We independently evaluate each feature and its patched counterpart(s), through a set of experiments performed on the popular FuzzBench benchmarking service [31]. We mainly used the bug-based dataset but also included the coverage-based one to clarify some cases where only one dataset was insufficient to draw conclusions. This allowed us to study each aspect in terms of its direct effects on the fuzzing campaign, as captured by the number of bugs and increased coverage. While these are the two metrics that researchers have settled upon to evaluate the overall performances of a fuzzer, in this paper we argue that these two values are often insufficient to gain insights about the impact of a certain feature or internal parameter. In fact, in our experiments, we often found that these two metrics alone did not provide enough information to fully capture the subtle difference between different implementations, which would allow security researchers and fuzzers' developers to debug and fine-tune their tools.

Therefore, for some of our final remarks, we limit our takeaways to qualitative findings, just relying on what we can learn by looking at general metrics such as bugs and coverage, and losing the necessary precision to measure deeper consequences of using a particular technique.

Overall, we can split our findings into two main groups. The results in the first group show that some features commonly adopted by off-the-shelf fuzzers root their origin in pragmatical or historical reasons, rather than scientific ones. For instance, we observed that a simple *random* energy assignment policy is capable in many cases to outperform the default AFL's energy assignment scheme. Similarly, we found that splicing implemented as a stage was less effective than splicing as a mutation, even though this comes with the caveat that the generated testcases are possibly more complicated to debug, thus affecting the usability of the system.

The second set of findings confirms the effectiveness of some historical design attributes when compared to modern alternatives. This is the case for novelty search, one of the major, and often forgotten contributions of AFL. It regularly outperforms the use of other genetic algorithms in terms of discovered bugs.

To conclude, we believe that the main contribution of our paper is to show how even apparently minor aspects can impact, both positively or negatively, the performance and outcome of a fuzzing campaign. We hope that our evaluation can pave the way for more sound and complete comparisons so that security practitioners and researchers can refine their tools to obtain the best results from their efforts. Therefore, in the spirit of open science, we release all code and artifacts to reproduce the evaluations for this paper.

2 FUZZ TESTING

Fuzz testing, or *fuzzing*, is a popular vulnerability discovery technique that executes a target as often as possible, in quick succession. For each run, it mutates the input to trigger novel and potentially buggy program points in the target.

The first fuzzers appeared in the early '90 [32], primarily relying on some forms of *blackbox* testing. In this case, the fuzzer provided the Program Under Test (PUT) with randomly generated inputs, with crashes and error conditions as the only guidelines for the fuzzing campaign. Early blackbox fuzzers were ready-to-use tools, which did not require any specific domain knowledge of the target applications [1].

More advanced examples of blackbox fuzzers are funfuzz [2] and Peach [15], which take the structure information about the testcases into account for their mutations. However, limitations of such approaches are quite evident, e.g., even simple conditional statements can become hard to bypass. More importantly, even if a random mutation can bypass a condition, the fuzzer remains unaware of this fact, unless the mutated input causes an immediate crash of the application. Thus, the fuzzer cannot use this information to generate new inputs.

The lack of target introspection led researchers to seek novel ways to reason about the internals of the programs. Hence, two very distinct paradigms were introduced: *whitebox* and *greybox* fuzzing. Whitebox fuzzing [20] relies on complex instrumentation and code analysis to produce more “interesting” inputs at the price of introducing a non-negligible performance slowdown [43]. On the other hand, methodologies that aim to reach the performances of blackbox fuzzers and to drive their exploration by using only lightweight code instrumentation fall under the category of greybox approaches. In this case, the code injected in the PUT typically only serves to produce some form of *feedback* to the fuzzer. This information is used to evaluate the quality of a testcase and, therefore, to progressively mutate only the interesting inputs and discard those that are not informative, according to the metric that the *feedback* represents.

Initially, both whitebox and greybox fuzzers shared some research directions, as in the case of the detection of bugs that do not result in a crash. In this context, the introduction of the so-called sanitizers incredibly augmented the precision of the fuzzers to detect memory corruption bugs [42], undefined behaviors like integer overflows [4] and other more specific classes of bugs [21].

Despite the advances that improved the performance of the new generation of whitebox fuzzers [37], greybox approaches remain the leading technique to discover vulnerabilities in modern codebases. For instance, Google's

OSSFuzz [3] makes use of greybox fuzzing approaches to test and detect bugs in a large number of popular open-source projects.

With the adoption of greybox fuzzing as the de-facto standard for the industry, researchers started to propose several methodologies to refine every single component of a greybox fuzzer to improve the bug-finding capabilities and performances. For instance, a key problem is how to mutate the testcases to increase the chances of triggering new behavior in the target. Traditional uniform mutation strategies [56] only work properly for some types of input and some applications, those that perform binary format parsing. More recently, approaches like AFLSmart and Zest [33, 35] suggested focusing mutations on a higher-level structure rather than on the raw bytes, e.g., by introducing AST-like representations of the input. The community also introduced the concept of *grammar-awareness* to indicate a fuzzer's ability to mutate an input according to certain grammar rules [5, 44]. In the scope of testcases management, another line of research focused on testcase scheduling, to maximize the explored code by optimizing the selection of the inputs present in the corpus [13, 51].

Other research directions instead explored different instrumentation techniques to study better forms of feedback. A popular form of feedback, usually considered the de-facto standard in the fuzzing community, is *code coverage*. This approach rewards the fuzzer when a new target execution results in a different coverage value, computed over the control flow graph (CFG) of the target application. In general, we refer to this family of approaches as *coverage-guided* fuzzing techniques. Consequently, the community has proposed multiple ways to measure the coverage that a certain input produces in the PUT, such as *block coverage*, that rewards the fuzzer when it hits a new basic block, and *edge coverage*, that instead measures newly discovered edges inside the CFG. These mechanisms allow a fuzzer to keep only those testcases that result in new coverage, leading the fuzzing campaign to go deeper in the application code, thus increasing the chances of eventually reaching the location of a bug. This simple idea is at the base of many modern fuzz testing projects, such as AFL++ [18], LibAFL [19], and libfuzzer [27], even though, as we will describe in this paper, the actual implementation is project-specific and can have a relevant impact on the performance of the fuzzer.

Finally, by extending the concept of feedback-guided fuzzing, researchers have proposed new forms of feedback, in the attempt to reveal different program locations or states not easily reachable by traditional techniques [12, 50]. Alternative forms of feedback may evaluate the quality of a testcase without relying on code coverage, but according to other aspects of the execution [6, 16, 29, 34].

3 AMERICAN FUZZY LOP

American Fuzzy Lop is a mutational coverage-guided fuzzer with a suite of additional tools [55]. These include testcase- and corpus-minimizers, a fault-triggering allocator, and a file format analyzer. Its latest available version at the time of writing is the 2.57b [1], released in 2020, but the fuzzer is unmaintained by its original author since 2.52b [2], released in 2017.

In this section, we will discuss the inner working of the fuzzer, `afl-fuzz`, and the design choices behind it.

3.1 General Design

As stated by Zalewski in a technical whitepaper [59] written in 2016, the main design principles behind AFL are *speed*, *reliability*, and *ease of use*. While important, these metrics are no longer the predominant principles that drive recent research on fuzz testing. Instead, researchers now predominantly focus on the time required to uncover bugs and on

[1] https://github.com/google/AFL/releases/tag/v2.57b
[2] https://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

the amount of coverage reached. While some choices in AFL improve these two metrics, the principle of ease of use is often forgotten, even though it is the reason behind many aspects of AFL. For instance, the corpus is represented as a queue for ease of use: by making AFL mutate simpler testcases first, shallow crashing testcases will have only minor changes over the original, "human-friendly" testcases. In addition, the fuzzer keeps track of the parent testcases of each corpus entry, allowing the user to reconstruct the genealogy of each corpus entry or crashing testcase.

The actions of the fuzzer are divided into *stages* that correspond to several tasks applied on a single testcase taken from the queue. Users may configure the behavior of these stages in different ways, for instance by disabling the deterministic stage with the `-d` parameter [61]. The testcase delivery to the target program is performed via standard input or through a file. Finally, the target execution is controlled by using a *forkserver* [57], a mechanism that uses pipes to request copy-on-write clones of the target programs with `fork(2)` for each execution to avoid the overhead of `execve(2)`.

3.2 Coverage Feedback

The main difference between AFL and previous solutions is the code coverage of the target program used as feedback. Although not the first to introduce this approach [14, 46], AFL took coverage guidance to the next level with an effective evolutionary algorithm based on this feedback.

However, the coverage metric AFL uses is not a classic path coverage. In fact, like many symbolic executors [7], AFL aims at a trade-off between precision and path explosion. Therefore, instead of simple basic block coverage, it uses edge coverage augmented with counters (*hitcounts*) to track the number of times an edge was executed. According to Zalewski [59], the use of hitcount buckets allows AFL to effectively tackle the path explosion problem.

Implementation-wise, AFL keeps a shared bitmap between the target and the fuzzer of 64kb (a value chosen to match the L2 cache size at the time AFL was first developed) with each entry of one byte. When an edge is executed, the corresponding entry is incremented by 1, wrapping around the byte in case of overflow. The instrumentation is at the level of basic blocks, so the ID used for each edge is the result of a hash function that combines the current block with the previous. This approach introduces collisions in the bitmap. Starting from version 2.37b (released in 2017), AFL adopted the `trace-pc-guard` option of SanitizerCoverage [26] for source-based instrumentation, an approximation of edge coverage that uses precise block coverage after breaking critical edges. After each traced execution, AFL post-processes the map and buckets the entries, thus reducing the possible values from 256 to 9. This mechanism is at the core of many fuzzers derived from AFL, such as AFL++ [18] and LibFuzzer [27].

This coverage information is used in the fuzzer by different algorithms. Its most important use is to decide if a testcase is *interesting*, and, therefore, whether it is worth adding it to the corpus for future mutations. For this, AFL uses a *novelty search* algorithm that considers as interesting an input that uncovers a new entry in the map or a value that reaches a previously unseen bucket.

The use of hitcounts allows AFL to encode each possible bucket as a bit in a single byte. Thanks to this optimization, AFL implements a very fast novelty search by using only a loop of DWORD/QWORD bit-wise operations. The choice of using 8 buckets allows AFL to avoid a path explosion, and, at the same time, increases execution speed, as it allows for a highly optimized processing of the resulting coverage map.

3.3 Scheduling

Like many other fuzzers, AFL makes use of multiple scheduling policies for various components.

First, it schedules which testcase in the corpus should be selected next. As described before, the corpus is represented as a queue and the base policy is FIFO. On top of that, AFL uses heuristics to decide to skip a testcase for various reasons. The first applies when there are some *favored* testcases in the corpus. The fuzzer marks a subset of the corpus as favored in the process of re-evaluating the queue and choosing a small subset of testcases that cover all the coverage seen so far, the so-called *corpus culling*. The main purpose of this operation is to give priority to testcases that are smaller and faster to execute. If there is at least one corpus entry in the favored set, a non-favored testcase is skipped with a 99% probability. Otherwise, the probability goes down to 95% in the case of a non-favored, but previously fuzzed entry, and 75% for never selected cases.

Another scheduling application is the so-called *energy assignment* [10]. For each corpus entry, AFL calculates a score that is used to compute how many executions must be performed in each stage in which mutator is used. The policy employed in the fuzzer, implemented in the `calculate_score` routine, is based on several parameters. The first is the execution time of the testcase, which can alter the score if slower, or faster than the global average from 0.1x up to 3x. Another parameter is the number of filled entries in the coverage map when executing the testcase, this time applying a multiplier from 0.25x to 3x. The intuition is that testcases with greater coverage trigger more interesting states. Additionally, the score is increased for newly discovered entries to allow the fuzzer to focus on novelties. Following the same spirit, the *depth* of the entry in the genealogical tree is taken into account as a multiplier to fuzz derived inputs, that could have been difficult to discover by blackbox approaches, for a longer time.

3.4 Mutators

AFL relies on generic, target-agnostic, byte-level mutators [56]. These are used in several stages, many of which are deterministic. The fuzzer sequentially bitflips the current input starting from one to 32 bits at a time. During this process, as optimization, AFL records the bits that do not contribute to a change in coverage to avoid mutating them in subsequent deterministic stages. After that, the fuzzer walks each byte by adding and subtracting integers in the range from -35 to +35. The next stage is the replacement of each part of the input with numbers from a set of interesting values, such as INT_MAX, 0, and 1. This is done iteratively on the input first at the byte level, and then by using 16 and 32 bits integers.

The last of the deterministic stages uses a dictionary [58] of tokens related to the input format, for instance, `\x7fELF` if the target is an ELF parser. These tokens can be specified by the user (with the `-x` parameter) to help the fuzzer to generate testcases that are otherwise impossible to create by using generic bit-level mutations. AFL can also auto-detect tokens during the bit flips stage by looking for groups of bits that, when changed, always produce the same coverage, a sign that they might be part of a magic value. The dictionary stages then mutates the testcases, replacing and inserting tokens from both, the user-specified and the generated list.

The first non-deterministic mutation stage is *random havoc*. It applies several mutations, including the ones used during the previous stages and some block-based mutations such as overwriting and inserting blocks of inputs. The mutations are applied at random locations of the input and are stacked. The number of applied mutations is chosen at random between 2 and 128 and the iteration of the stage is regulated by using the score of the testcase.

The last stage, *splicing*, by default is activated only after the fuzzer goes through a full cycle of the entire queue without any new finding (but it is always enabled in FidgetyAFL [61]). It selects an entry from the corpus and recombines it with the current testcase, then it applies the havoc mutator to this child testcase. This is an important stage that allows AFL to generate testcases derived from two parents.

In AFL, for ease of use, each testcase saved in the corpus or the crash folder keeps the information about its one or two parent testcases, as well as the mutations that were applied. This allows AFL's users to reconstruct the entire process of derivation of a testcase, information that helps them during crash analysis.

3.5 Minimization

Some mutations can increase the size of a testcase and, especially for inputs discovered later in a testing campaign, can result in very lage files. These large, slow-to-parse, inputs can decrease execution speed and decrease the likelyhood of a mutation of the correct bytes. Therefore, the fuzzer tries to minimize their impact by using a testcase minimization algorithm.

After requesting a testcase from the corpus, AFL passes it through its *trimming* stage. The key idea is to mutate the testcase by trying to obtain a smaller testcase that still achieves the very same coverage. The algorithm consists of removing blocks from the inputs while checking if the coverage map remains the same. If successful, the process is repeated several times by increasing the size of the blocks to remove. This technique reduces the complexity of the items in the corpus, but it also requires additional executions for each testcase that is saved in the queue.

3.6 Instrumentation

To obtain the coverage information from each execution of the target, AFL employs several instrumentation options. First of all, it can instrument the x86 compiler to intercept and modify the assembly code, to log each basic block by relying on functions available through an injected runtime. In addition, AFL also provides an LLVM pass [25] which assigns a random block ID at compile time and adds the instrumentation to hash the blocks and write to the shared memory, thus resulting in a more efficient instrumentation than the one provided by the legacy x86-only solution. With LLVM, the runtime is also more mature as it provides not only the forkserver option but also the so-called *persistent mode* to avoid forking when fuzzing stateless code, resulting in increased performance.

Alongside compiler-based approaches, AFL comes with a binary-only QEMU mode. QEMU mode uses a patched QEMU 2.10 usermode to inject a forkserver at the guest entrypoint, and to add instrumentation between each executed basic block, through a logger routine executed after each basic block.

4 METHODOLOGY AND EXPERIMENTS DESIGN

By reviewing the implementation and the internals of AFL, we identified nine characteristics to assess in our tests. For each of them, we also looked for alternative solutions proposed in other works to serve as a comparison in our experiments. We have not selected the trivial comparison between AFL and FidgetyAFL [61] as it is covered in the FuzzBench paper [31], which highlights that FidgetyAFL always outperforms AFL in terms of code coverage over time.

Our aim is to assess the contribution of each feature on the performance of AFL in terms of uncovered bugs and code coverage using FuzzBench [31] over a 23h campaign. If the results depend highly on the structure of the target program, we will try to classify manually which kind of program is influenced by the tested feature. Finally, when the results do not show a significant difference, we will provide a qualitative investigation of the possible impact of that feature on usability.

We now introduce the nine aspects to be covered in our study.

Hitcounts. Hitcounts are adopted by other fuzzers today [27, 47], but AFL was the first to introduce this concept. Despite its wide adoption, the impact of this optimization (over plain edge coverage) has never been measured in isolation on a large set of targets.

To fill this gap, we modified AFL not to increase each entry in the coverage map while instrumenting the target. Instead, we always set the value to 1. We expect hitcounts to improve the coverage and especially the bug detection capabilities by introducing additional information about the program state, like loop counts. We want to quantify this improvement and potentially discover target-specific corner cases.

Novelty search vs. maximization of fitness. While AFL considers every newly discovered hitcount as interesting, both, other early fuzzing solutions [60], and more recent tools [38] instead only consider testcases that maximize a given metric as interesting. For instance, VUZZER uses the sum of all the weights of the executed basic blocks [38].

We think that a big part of the success of AFL in terms of performance is the novelty search-based approach it uses to evaluate interesting testcases. To evaluate this assumption, we implemented [3] a simplified version of the VUZZER fitness maximization without the need for static analysis, in which each basic block has weight 1:

$$f(i) = |\mathrm{BB}(i)| \begin{cases} \frac{\sum_{b \in \mathrm{BB}(i)} \log_2(\mathrm{freq}(b))}{\log_2(\mathrm{len}(i))} & \text{if } \mathrm{len}(i) > 50000 \\ \sum\limits_{b \in \mathrm{BB}(i)} \log_2(\mathrm{freq}(b)) & \text{otherwise} \end{cases}$$

We chose to borrow the VUZZER fitness function as it is a simple one based on just code coverage, avoiding introducing a fitness from scratch as, to the best of our knowledge, VUZZER is the only academic work proposing a simple fitness. Other approaches in the literature that use a fitness employ heavy static analysis or complex approaches based on many features, not just code coverage [30]. While it would be interesting to benchmark them too, it is not fair to compare such complex techniques with a fuzzer that only uses code coverage like AFL. More complex novelty search solutions are present in literature [52] that can be used as a competing approach in future works.

In this experiment, we benchmark the AFL approach versus a fitness maximization and the combination of the two approaches, as proposed by VUZZER [38]. We expect novelty search to outperform both of the competing algorithms, as the maximization saves testcases in the corpus that are not small and fast (one of the key elements in the design of AFL), and might end up in local maxima. A set of diverse testcases, like the ones saved by AFL, is likely better in the corpus during fuzzing.

Corpus culling. The prioritization of small and fast testcases in the AFL corpus selection algorithm improves the speed at the cost of avoiding more complex testcases that might trigger more complex program states. We selected this feature for our benchmark because the set of *favored* testcases in AFL was a major addition to the fuzzing algorithm, and it is used even as a metric in following works such as Driller [45].

In this experiment, we want to assess the difference between using the AFL corpus culling mechanism and using the entire corpus. We expect culling to result in faster coverage growth over time and, potentially, more bugs triggered in the same time window. On the other hand, the fuzzer without corpus culling might be able to discover new bugs that standard AFL is unable to trigger.

[3] Note that the input length is bound to 50,000 bytes (to address input bloating) and the log base is taken from the VUZZER code.

Score calculation. The performance score used to calculate how many times to mutate and execute the input in the havoc and splice stages are derived from many variables, mainly testcase size and execution time. This score is an essential part of AFL and the focus on many derived works (e.g. [9, 10, 54]).

In our experiment, we measure the difference between the AFL solution and two baselines, represented by a constant and by random scores. As picking a constant is a sensitive operation, we opted to create two AFL variants, one with the minimum score possible for AFL (25) and another with the maximum (1600). The random variant selects instead a random number within these boundaries. In addition, we include in the experiment a version of `calculate_score` that does not prioritize novel corpus entries, as this was a significant optimization introduced in AFL. Intuitively, we expect that the major contribution comes from the prioritization of the novelties, thus resulting in small differences between the baselines and the patched AFL with the naive score calculation.

Corpus scheduling. The FIFO policy used by AFL is only one of the possible policies that a fuzzer can adopt, to select the next testcase. However, derived works tend to take for granted that the corpus structure is represented by a queue.

While we know that this feature has its root in usability, in this experiment we assess whether it also contributes to the performance of the fuzzer. Thus, we evaluate AFL versus a modified version that implements the baseline (i.e., random selection) and the opposite approach (i.e., a LIFO scheduler). We expect the random approach to perform equal to, or even better than the original embodiment of AFL, while the LIFO approach may help in gaining coverage faster on some targets.

Splicing as stage vs. splicing as mutation. Splicing refers to the operation that merges two different testcases into a new one. There are two possible ways to apply this mechanism. The first, adopted by AFL, considers splicing as a stage. In this case, the actual merge happens only once at some point in the execution of a specific testcase, when it is joined with a randomly chosen input among the other ones present in the queue. However, other fuzzers (e.g., Libfuzzer [27]) often implement splicing as a mutation rather than a stage, thus applying it many more times for each testcase during their havoc stage.

To compare the two solutions, we modified the AFL codebase to implement splicing as a mutation operator. This choice can also have an impact on the usability of the fuzzer. Indeed, we expect that a major adoption of splicing as mutation can increase the exploration of the fuzzer while reducing the simplicity of the testcases and, therefore, complicating the a-posteriori triaging phase.

Trimming. Trimming testcases allows the fuzzer to reduce their size and consequently give priority to small inputs, under the assumption that large inputs slow down the execution and that the mutations would be less likely to modify an important portion of the binary structure. In AFL, the component in charge of this task tries to discard blocks of data with variable length and stepover. When the removal results in the same checksum of the original trace map, the new minimized testcase is stored.

Even though this algorithm can bring the two important benefits described above, we argue that reducing the size of the testcases could reduce state coverage. Additionally, the trimming phase could become a bottleneck for slow targets. Therefore, in our evaluation, we compare the default version of AFL against a modified one, in which we disabled trimming. We expect trimming to be either beneficial or detrimental, depending on the type of target program and the structure of its input.

Timeout. The timeout regulates the maximum amount of time the target program runs for. This greatly influences the execution time of the target and in turn the number of executions per second. While the user can specify an arbitrary value by passing a command line argument (`-t`), AFL can also automatically compute a timeout for the program under test. More specifically, as a first step, AFL calibrates the execution speed during an initial phase by running the target several times and computing an average of the execution times. After that, the default heuristic applies a constant factor (x5) to this average value and rounds it up to 20 ms. In our experiments, we modify the multiplicative factor to measure its effect on the fuzzing session. We expect that a higher timeout can lead to higher coverage, but also degrade the performance of the fuzzer.

Collisions. As explained in section 3.6, AFL assigns an identifier for each basic block at compile-time. When using SanitizerCoverage [26]'s *pcguard*, critical edges are split into basic blocks and thus AFL assigns a random identifier to each edge. Unlike the classic instrumentation that combines the IDs of the current and the previous block, however, this technique is unable to track edges related to indirect jumps. For both variants, since identifiers are chosen at random, this causes collisions between two different edges in the bitmap, that in turn can affect the novelty of a testcase. Although the number of collisions depends on the number of instrumented locations, for an average size program the actual collisions are typically between 750 and 18,000 [23].

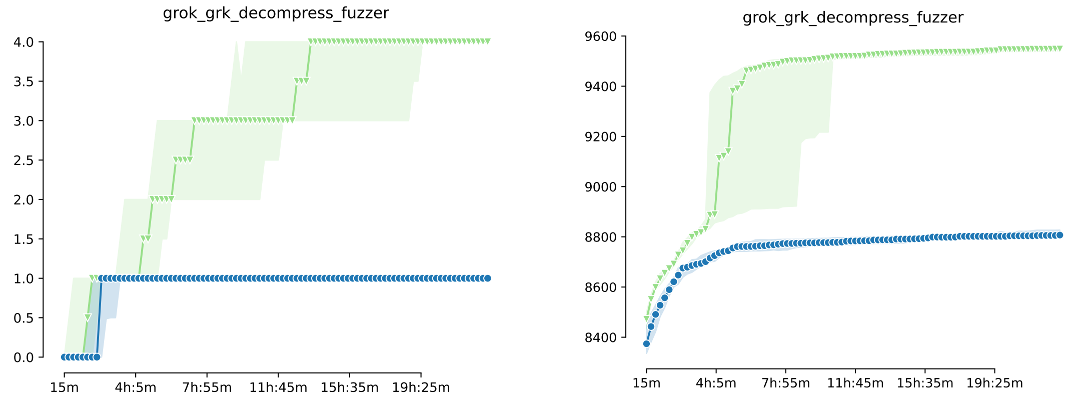
In our evaluation, we want to compare the AFL instrumentation approach against a version that is collision-free. As SanitizerCoverage traces each block by calling a function with a guard parameter, and this guard is contained in a per-module table initialized in a constructor, we can easily patch AFL to assign values to the guards by using a global incremental counter in the constructor instead of random values. This allows the instrumentation to generate edge encodings that do not result in collisions during the fuzzing session. Other fuzzers indeed make use of the guard variable as the index to access the fuzzer bitmap.

We want to benchmark this feature as the collision-free variant is simpler than the original implementation with *pcguard*, raising the question of why random identifiers are even used in AFL. In addition, it is unclear if the lack of feedback from the indirect jumps affects the performance more than the collisions, so we include in our test also the classic approach to benchmark this impact.

Please note that in this experiment, unlike the collision-free coverage based on *pcguard* present in AFL++ (since 2.66c), we do not adapt the size of the map to the detected number of blocks – a feature that significantly improves the speed of the fuzzer – as we want to evaluate the impact of the collisions in isolation.

5 EXPERIMENTS

In this section, we present the results of our experiments, conducted by using the FuzzBench service [31], and we discuss them to understand the impact of each selected feature. We mainly use the bug benchmark of FuzzBench, which consists of 25 targets known to contain bugs, as we believe that discovered bugs are the ultimate metric in fuzzing evaluation [24]. In addition, we also report the coverage over time as another important metric to understand the performance of each variant of AFL. Each program was executed for 23 hours and the reported results are median values computed over 20 trials to mitigate the effects of randomness in fuzzing. In addition, we use the Mann-Whitney U test to verify the statistical significance of the results by comparing differences between two independent groups that in our case are the original AFL and its variants. The aggregation of the results is done by using an average normalized score [31]. Finally, we executed all variants with the `trace-pc-guard` instrumentation and persistent mode to mitigate the well-known impact [53] of `fork(2)`.



(a) Bug Detection

(b) Coverage

Fig. 1. Comparison of AFL and AFL-edge-coverage on Grok grk decompress (🟦 `AFL`, 🟩 `AFL edge coverage`)

Table 1. Hitcounts vs. plain edge coverage bug-based experiment score

Fuzzer	**Average normalized score**
`AFL edge coverage`	88.09
`AFL`	74.36

For the sake of brevity, we only report the results of interesting benchmarks and avoid discussing each individual benchmark for each experiment. For the interested reader, the graphs with the complete data of all the 9 experiments are available online at `https://anon-afl.github.io/dissecting_afl_reports/`.

For each set of experiments, we also highlight in gray our discovered insights. We hope this can help users to better understand AFL and improve the design of new fuzzing approaches.

5.1 Hitcounts

In this first set of experiments, we compare vanilla AFL against a modified version that does not use hitcounts. Table 1 reports the average normalized score of the number of uncovered bugs in our experiments. Quite surprisingly, the AFL variant without hitcounts discovered more bugs than the unmodified AFL, a counter-intuitive result as hitcounts should allow AFL to bypass coverage roadblocks that depend on loop counts.

In particular, vanilla AFL performed better on 6/25 benchmarks in terms of median discovered bugs, out of which only two are statistically significant for the Mann-Whitney U test. The variant with only edge coverage was better on 5/25 benchmarks, of which four are statistically significant.

It is interesting to note how for some targets edge coverage clearly outperformed vanilla AFL, as in the case of the grok and the PHP benchmarks. For instance, in the case of `grok_grk_decompress_fuzzer` we can observe that the graphs reporting bugs discovered over time and coverage over time (Figure 1) are correlated. This might suggest that the use of hitcounts prevents the fuzzer from discovering new code paths, a behavior that can be explained by the augmented sensitivity, up to 8x as the hitcounts introduce 8 different states for each edge.

As shown by previous studies [16, 50, 51], the increase of sensitivity introduces testcases in the saved corpus that are too similar to one another, causing internal wastage of the exploration of the program. AFL is therefore focusing on fuzzing testcases that are not frontiers in terms of unexplored coverage areas. This behavior is, of course, highly target dependent, as the states that AFL can reach by using the hitcounts in its feedback may contain bugs that otherwise cannot be easily discovered with edge coverage only.

Table 2. Hitcounts vs. plain edge coverage code coverage-based experiment score

Fuzzer	Average normalized score
`AFL`	99.63
`AFL edge coverage`	97.99

To further confirm our intuition that hitcounts introduce a benefit only on some targets, we run another set of experiments on FuzzBench on a different set of 22 benchmarks that FuzzBench uses to evaluate fuzzers using only code coverage as a metric[4]. The score reported in Table 2 shows that on this set of different subjects classic AFL outperforms the variant with only edge coverage, confirming that hitcounts can either increase or decrease the effectiveness of the fuzzer depending on the target application.

Our conclusion after this experiment is that AFL, and follow-ups fuzzers like AFL++, should provide an option to disable hitcounts. AFL++ provides many different options, and the users are suggested to run an instance of each variant when doing parallel fuzzing, a common use-case in real-world setups. The fact that in our experiments, hitcounts have shown very different results on different targets suggests that users should include a variant without hitcounts when doing parallel or ensemble fuzzing like OSS-Fuzz [3].

5.2 Novelty search vs. maximization of a fitness

In this second experiment, we compare three fuzzers: vanilla AFL (with its novelty search algorithm), a variant with only fitness maximization, and a hybrid variant with both maximization and novelty search. In line with our expectations, the bug-based benchmark shows that, in average, vanilla AFL performs best. Table 3 reports the average normalized score of the number of uncovered bugs.

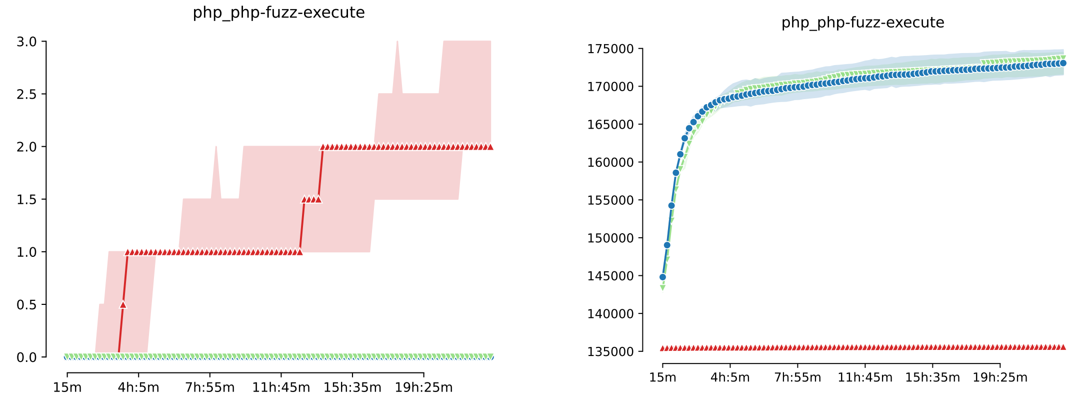
The usage of the fitness only is clearly detrimental and the combination of both techniques does not introduce a valuable increment in bug-discovery. AFL and the combined variant perform almost the same, with the exception of `libhtp_fuzz_htp` in which the fitness variant is better and `poppler_pdf_fuzzer`, in which vanilla AFL is best. While this result was expected, there are some surprising results on specific targets such as `php_php-fuzz-execute`, placing the variant with only the fitness maximization as the best fuzzer on 4/25 benchmarks, all statistically significant.

Unlike in the previous experiment, this time there is also no correlation between uncovered code and bugs. For instance, Figure 2 shows that for PHP the variants with fitness only are unable to increase the coverage of the target application, but at the same time, it is the only variant able to discover bugs. The saved testcases in the corpus cover the same regions as the initial testcases so we can observe that, on this target, the fuzzer is behaving like a blackbox fuzzer without any coverage tracking capability.

[4] https://www.fuzzbench.com/reports/experimental/2021-12-17-afl-edges/index.html

Table 3. Novelty search vs. maximization of a fitness bug-based experiment score

Fuzzer	Average normalized score
AFL	83.32
AFL fitness	83.08
AFL fitness only	70.17



(a) Bug Detection

(b) Code Coverage

Fig. 2. Novelty search vs. fitness experiment on the PHP application (🟦 AFL, 🟩 AFL + fitness, 🟥 AFL fitness only)

A possible explanation is that the novelty search fuzzers are spending time exploring more program behaviors, while the bugs are in the initial code regions behind constraints that cannot be solved immediately. On large programs, this is a well-known behaviour [11] which explains why random testing can outperform more complex solutions.

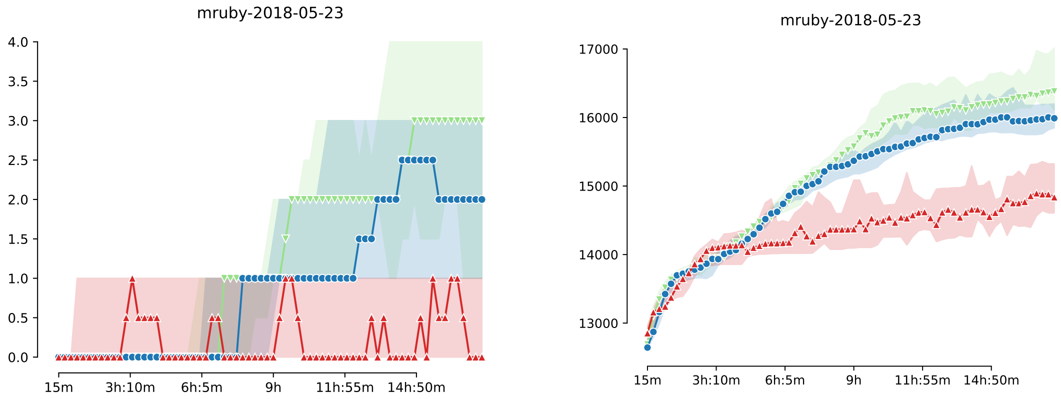
The conclusion we can draw from this experiment is that it would be a mistake to underestimate the impact of the novelty search. In particular, researchers proposing new approaches that also modify this aspect should carefully evaluate – in isolation – the benefit of a different mechanism to decide if an input is interesting, as AFL's novelty search provides a strong baseline.

5.3 Corpus culling

In this third experiment, we evaluate AFL versus two other variants, one without corpus culling and one with culling without any prioritization based on the `fav_factor` (a function of execution speed and testcase size). In Table 4 we report the average normalized score for each fuzzer in terms of discovered bugs.

Table 4. Corpus culling experiment score

Fuzzer	Average normalized score
AFL w/o fav_factor	90.14
AFL	87.00
AFL no culling	81.94



(a) Bug Detection

(b) Code Coverage

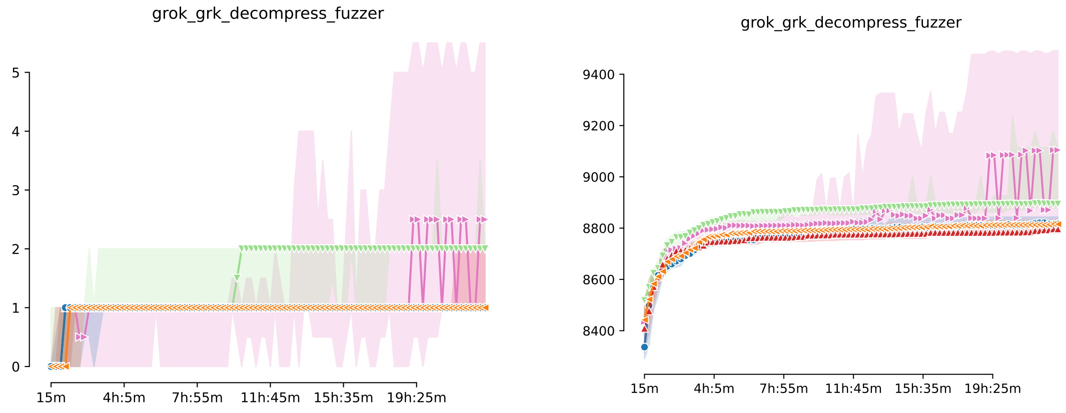
Fig. 3. Corpus culling comparison on the mruby application (🟦 `AFL`, 🟩 `AFL w/o fav_factor`, 🟥 `AFL no culling`)

The usage of corpus culling is clearly a benefit looking at the total numbers, but the prioritization given by using `fav_factor` as weight is not. The reason behind this is that corpus culling favors faster testcases while maintaining the same code coverage, but loses the state triggered by more complex testcases that cannot be easily observed by looking at edge coverage alone. While this optimization is effective as it reduces the number of testcases in the queue, prioritizing always the smaller and faster inputs may be decremental in findings bugs. In fact, our experiments show that the more naive version of culling seems to be the most effective in practice.

Taking `mruby-2018-05-23` as case study (Figure 3), the variant without the fav_factor provide the best results. The graphs show how the exploration of the program states not related to code coverage can help the fuzzer to increase coverage faster. While this may seem counter-intuitive, there are programs states blind to edge coverage (e.g. loop counter values) that are roadblocks in terms of exploration of the control flow graph. Therefore, fuzzing more complex testcases can help to bypass them and achieve new coverage. While this variant is generally better, the results are only statistically significant for the `mruby` and the `openh264_decoder_fuzzer` applications.

Investigating the results of the variant without culling also provides interesting insights. The outcome is highly variable, with benchmarks like `php_php-fuzz-execute` and `poppler_pdf_fuzzer` in which it discover more bugs and more code coverage in a statistically significant way, and others like `grok_grk_decompress_fuzzer` in which it is only able to discover two bugs while the others discover six. This can be explained by looking at the number of testcases in the queue, as the fuzzer got stuck fuzzing testcases too similar to one another if culling is disabled.

Our experiments show that complex testcases are useful to uncover bugs and AFL should not discard them a priori. However, it is unclear how to reach the right trade-off between complexity and speed and we foresee future works that try to improve the prioritization algorithm of corpus culling to fuzz faster without discarding interesting testcases.



(a) Bug Detection

(b) Code Coverage

Fig. 4. Score computation comparison on grok (🟦 `AFL`, 🟩 `Max`, 🟥 `Min`, 🟪 `Random`, 🟧 `No novel`)

5.4 Score calculation

To evaluate the algorithm used in AFL to assign the energy to a testcase, we compare it versus several alternative implementations: two simply adopting a constant score (respectively the maximum and the minimum possible score in AFL), and one that assigns a random score between the valid range. Additionally, we include a variant of the AFL scoring algorithm without the multiplicative factor that prioritize the novel inputs recently saved in the queue.

Table 5. Score calculation experiment score

Fuzzer	**Average normalized score**
`AFL random score`	93.52
`AFL`	90.51
`AFL max score`	88.88
`AFL no novel`	87.63
`AFL min score`	81.06

In our tests, vanilla AFL was not able to outperform the random baseline in terms of an average normalized score of uncovered bugs (Table 5), while it was better than the variants using a constant score. The results confirm instead that the prioritization of novel testcases is an improvement.

However, we can observe that the normalized scores of AFL and the random variants are really close and the random version is the best performer only in three benchmarks (`arrow_parquet-arrow-fuzz`, `grok_grk_decompress_fuzzer`, and `php_php-fuzz-execute`), out of which only the results on PHP are statistically significant.

The high variability of the random score fuzzer can be observed for instance when testing `grok`. Figure 4 shows that in some runs this fuzzer outperformed all other variants by a large margin (both according to bugs and coverage), but in other runs, it did not.

It is worth observing in this case that the high variability in code coverage (Figure 9b) is always above the curve of the other variants, suggesting that the random fuzzer consistently outperforms the others while the number of bugs is more aleatory. This highlights that uncovering a bug is more susceptible to randomness.

The result of the random variant is particularly important as in recent years energy assignment was the focus of a large number of studies, most of which used AFL as a baseline for the experiments. However, if even a random score can often perform better than the algorithm implemented in AFL, it is difficult to say whether a new energy assignment algorithm that beats AFL is really an improvement that can increase the ability of the fuzzer to discover bugs if not compared against the real baseline. The high variance in the results of the random algorithm also suggests that it might be a useful adoption in parallel fuzzing. Multiple instances of the fuzzer using this score calculation algorithm will increase the chance to hit the best performer random distribution.

A possible threat to the validity of this experiment is the biased nature of the benchmarks, which contain applications with a medium-small sized codebase as they are libraries. With complex targets, the score calculations with a non-naive algorithm may become more important, and we can see a hint of this result by looking at the results of this experiment for the `ffmpeg_ffmpeg_demuxer_fuzzer`, a complex and slow program in which AFL triggers more median number of bugs than the random variant.

In conclusion, our experiments show that for simpler targets the energy assignment problem may be less important than for complex programs. On the one hand, this might suggest that the development effort in creating faster and more effective fuzzers can make this allocation problem less relevant for generic fuzzers. On the other hand, the fastest possible fuzzer cannot compensate for a slow and complex to execute system under test (like program interpreters or even entire operating systems), highlighting the need to benchmark new energy assignment algorithms, with a dataset of complex targets. Finally, we suggest using the baselines we introduced in this paper to avoid the mistake of considering AFL's implementation as a baseline.

5.5 Corpus scheduling

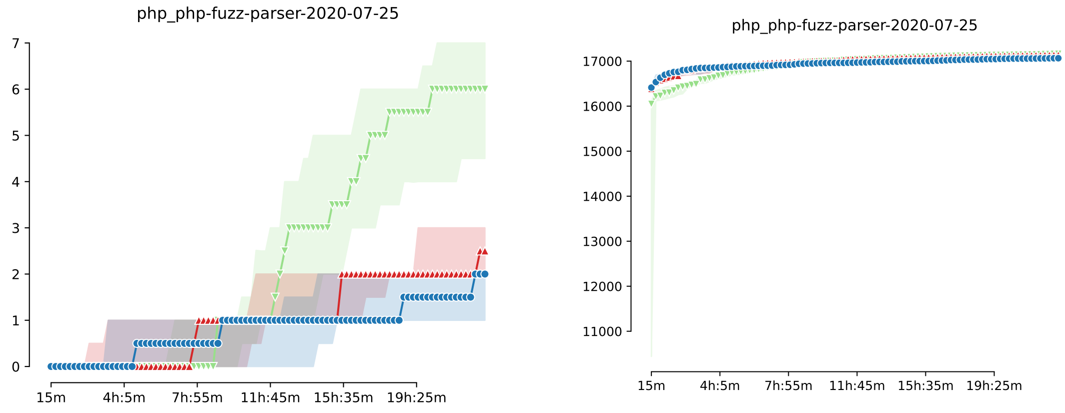
In this experiment, we tested two alternatives to the FIFO policy used in AFL to select the next testcase in the queue to fuzz: a random selection, and a LIFO policy. The results show that in terms of ability to discover bugs, vanilla AFL is better than the random baseline, but the LIFO variant is the best among the three, as reported in Table 6.

Table 6. Corpus scheduling experiment score

Fuzzer	Average normalized score
`AFL scheduling LIFO`	90.33
`AFL`	82.94
`AFL scheduling random`	82.94

This time, the random baseline is not superior to vanilla AFL, which is better than random in 6 benchmarks, making the results of the previous works in this field, corpus scheduling or *seed scheduling*, robust even if they use AFL as baseline. However, a simple variation such as LIFO, shows a boost in performance on 7 targets, two of them in a statistically significant way.

For instance, on `php_php-fuzz-parser-2020-07-25` (Figure 5), fuzzing later discovered testcases first with FIFO gives a boost in the bug discovery ability while maintaining the uncovered coverage regions at the same level of the other AFL variants. Another benchmark that benefited from the LIFO approach is `grok_grk_decompress_fuzzer`, but in this case the performance boost is affect both bugs and code coverage.



(a) Bug Detection

(b) Code Coverage

Fig. 5. PHP fuzz-parser for the corpus scheduling experiment (🟦 AFL, 🟩 LIFO, 🟥 Random)

On the other hand, when the scheduling policy decreases code coverage (like in `matio_matio_fuzzer` and `mruby-2018-05-23`), it negatively affects the results and performs worse than the vanilla AFL and the random baseline. The improvement of LIFO over AFL seems to not be related to the type of the input, for instance, it performs better on the PHP benchmarks but worst on `mruby`, both of which are textual programming languages parsers. Thus, we were not able to reach a clear conclusion on which policy is better in general, as fuzzing the newly generated testcases first is not always the best choice even if LIFO is the top performer on average.

However, the boost in performance can be easily observed once that LIFO starts diverging from AFL, making possible to learn during the fuzzing campaign which policy is best suited for a given target, even with a simple approach that alternates between the two in the first hours and then select the best performer.

In conclusion, our experiments show that FIFO is generally better than random, not just because of usability but also because it often provides better results. However, on many targets, the alternative approach (LIFO) provided better results. Thus, we believe that more research is needed to learn the best policy for corpus scheduling (such as the recent AFL-Hier [51]). Even if the random baseline seems weaker than AFL, we believe future works on the topic should still include it as a baseline in their evaluation alongside other simple policies like FIFO and LIFO.

5.6 Splicing

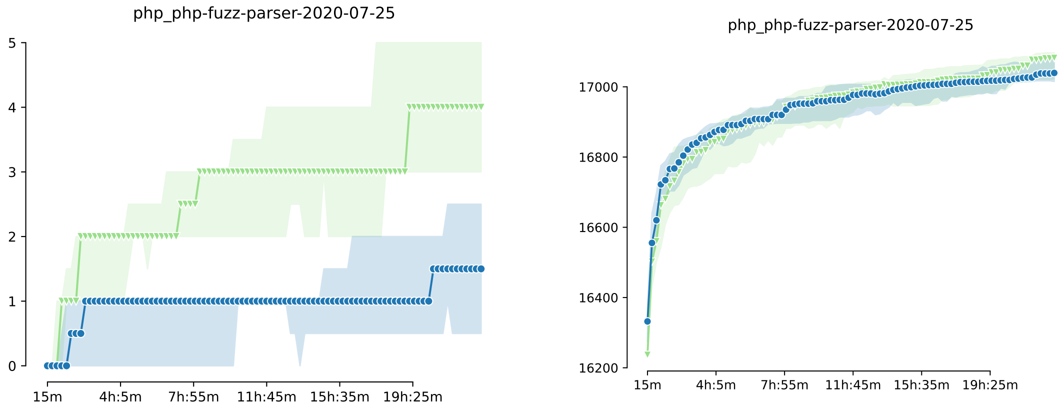
In this experiment, we evaluate AFL splicing stage (in which the currently fuzzed testcase is combined with another taken from the corpus, and then fuzzed with the havoc stage) versus a variant in which the combination of two or more testcases is implemented as one of the mutations included in the havoc stage. Table 7 reports the average normalized score in terms of uncovered bugs for this experiment.

The results show that the AFL variant that uses splicing as part of the havoc stage outperforms vanilla AFL. In particular, it is better in 6 benchmarks – two of which show a statistically significant difference from AFL.

In this experiment, the insight is clear, the recombination of different inputs in the fuzzer corpus leads to a benefit especially on highly structured inputs parsers such as `php_php-fuzz-parser-2020-07-25`, as shown in Figure 6, in

Table 7. Corpus scheduling experiment score

Fuzzer	Average normalized score
`AFL splicing mutation`	97.15
`AFL`	94.66



(a) Bug Detection

(b) Code Coverage

Fig. 6. Splicing Comparison on `libxml2` (🟦 `AFL`, 🟩 `AFL Splicing Mutation`)

which the improvement in terms of bugs finding is very large. Even on other benchmarks in which the difference in terms of bugs is smaller, like `mruby-2018-05-23`, splicing as a mutation improved code coverage. This result is not surprising, as recent works [8, 17] have already shown that an enhanced mutator that can recombine and merge different inputs in the corpus provides a net benefit for a fuzzer testing highly structured inputs parsers.

The downside is that frequent re-combination increases the testcase complexity, making triaging harder. In fact, by introducing splicing as mutation we lose the ability of AFL to keep track of which testcases were involved in the recombination by simply looking at the file names. Moreover, with this new variant, the number of parent nodes for each testcase can be greater than two, thus reducing the ability of users to easily keep track of the transformations applied by the fuzzer.

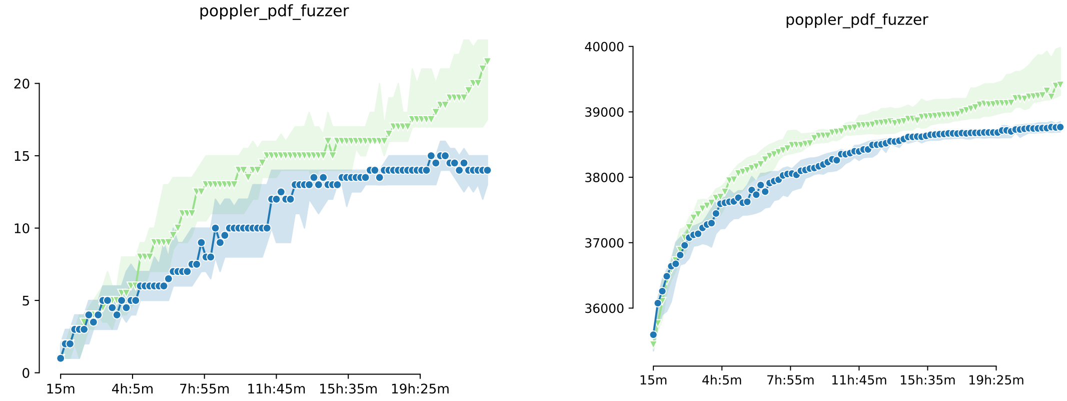
The results of this experiment confirm our intuition about the effectiveness of splicing. Modern fuzzers focus on performance as bugs are becoming harder and harder to find, and therefore we believe they should use splicing as a mutation. In the specific case of splicing, both versions can co-exist in the same fuzzer without conflicts and enabled according to user preferences.

5.7 Trimming

In this experiment, the comparison is between AFL (which uses trimming to reduce the size of a testcase while maintaining the same code coverage) and a variant without the trim stage. The overall result in terms of average normalized score of the bugs found is reported in Table 8, which highlights that the variant without trim stage outperforms vanilla AFL.

Table 8. Trimming experiment score

Fuzzer	Average normalized score
`AFL no trim`	99.25
`AFL`	88.81



(a) Bug Detection

(b) Code Coverage

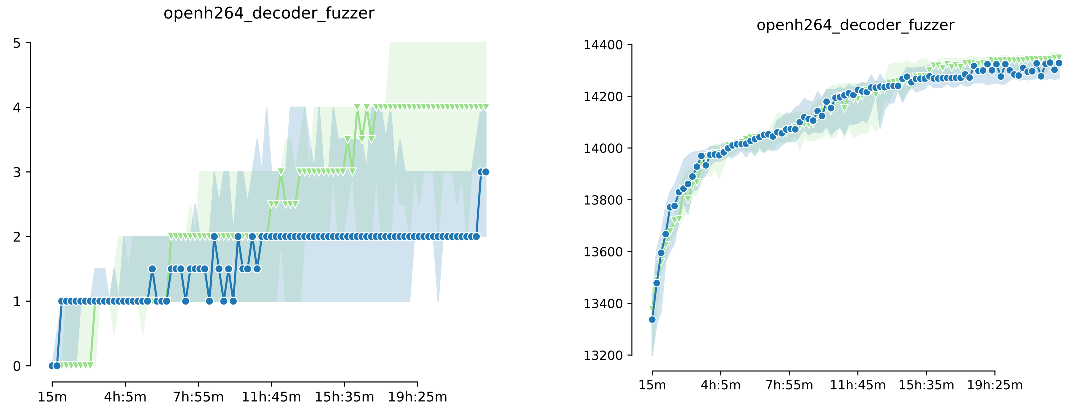
Fig. 7. Trimming Comparison on `poppler` (🟦 `AFL`, 🟩 `AFL no trim`)

The aim of the trim stage of to speedup the fuzzer by reducing the size of testcases while maintaining intact the coverage. This can negatively affect the effectiveness of a fuzzer in the cases in which code coverage alone is not sufficient to describe the program state. So it is not surprising that on many targets the trim stage decreased the performance of AFL.

In particular, the variant without trimming was the best performer against eight targets (five of which with statistical significant results). It is worth noting that all these five programs process structured inputs and therefore the trim stage may be detrimental. In fact, attempts to shrink the testcases would most likely result in invalid inputs, and therefore AFL ends up using valuable resources to try to trim the corpus, without actually succeeding.

As an example, the variant without trimming provided great results on a complex structured input program such as `poppler_pdf_fuzzer` (Figure 7) in which it can discover more bugs and explore more code without reaching saturation (as suggested by the flattening graph of AFL). The application parses PDF, a complex format in which bit-level modification can easily destroy the validity of an input resulting in different coverage and so a useless (and time-consuming) trimming stage. In addition, the longer the fuzzing campaign the slower it becomes to execute testcases, making trimming more and more costly without any advantage as the fuzzer is unable to alter a testcase without maintaining the same code coverage.

The insight from this experiment is that the trim stage can be useless or even detrimental when the tested codebase is large or when the target input has a complex format, as every bit-level modification will unlikely lead to a testcase that maintains the same coverage of the original. This behavior wastes resources, as the time spent



(a) Bug Detection

(b) Code Coverage

Fig. 8. Timeout comparison on `openh64decoder` (🟦 `AFL`, 🟩 `AFL double timeout`)

trimming (without any benefit) could be better used for fuzzing. Therefore, in these cases we believe that a fuzzer without the trim stage outperforms AFL simply because it fuzz the target at a higher speed.

5.8 Timeouts

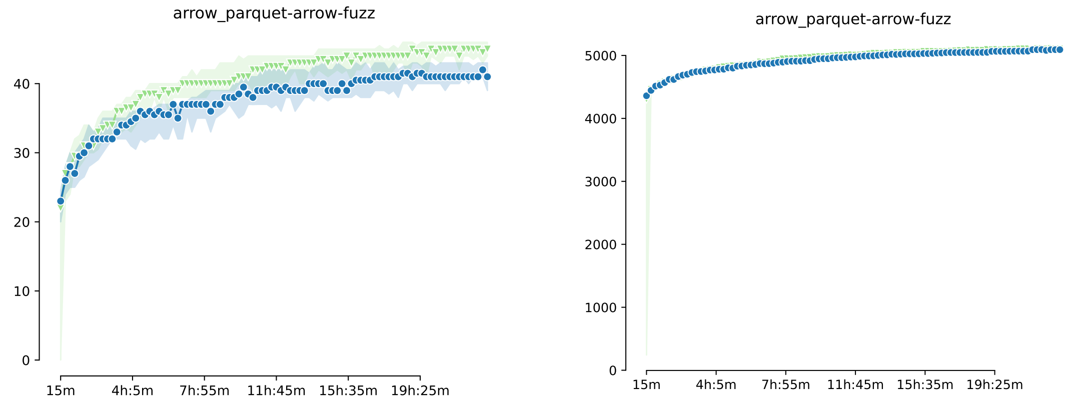
In this test we compare AFL with a variant that double the timeout chosen by the timeout detection algorithm. In terms of average normalized score, reported in Table 9, this variant seems to perform better than vanilla AFL.

Table 9. Trimming experiment score

Fuzzer	**Average normalized score**
`AFL double timeout`	97.43
`AFL`	94.70

The variant finds a higher median number of bugs than AFL in seven benchmarks and performs worst than AFL in two – but none of the results are statistically significant. However, we can see how on some targets, like `openh264_decoder_fuzzer` shown in Figure 8, doubling the computed timeout helps the fuzzer to find bugs faster. This small difference is due to the environment in which the fuzzers were run, a preemtible VM on the cloud, the default environment of the FuzzBench service. Usually, a fuzzer decreases its execution per second with time as more complex code paths are discovered, and slow targets are not an exception. A slow target like `openh264_decoder_fuzzer` on a slow machine may cause the fuzzer to generate inputs triggering timeouts virtually at every execution if the timeout is too strict.

This experiment, however, may not suggest a generally valid insight about AFL, as the speed of a target program depends on the complexity of such program but also on the method used to fuzz it. AFL in forkserver mode, for instance, may behave in a different way than the setup used in this experiment (which uses persistent mode), or other alternative



(a) Bug Detection

(b) Code Coverage

Fig. 9. Collisions comparison on `arrow` (🟦 `AFL`, 🟩 `AFL collisions free`)

solutions used by AFL-based fuzzers to execute a system under test (such as full-system fuzzers [22, 40] or network fuzzers [36, 41]).

In conclusion, this experiment suggests that changing the timeout calculation algorithm of AFL with a similar one does not change much the performance of the fuzzer. This aspect is too much dependent on the platform in which the fuzzer runs to get a generic insight and the user should carefully tune the timeout based on the slowdown introduced by using different types of machines or virtual environments, such as cloud VMs or containers.

5.9 Collisions

In the final experiment, we tested AFL versus a collision-free variant that uses the `trace-pc-guard` instrumentation option of LLVM to track edges in the target program. Table 10 shows that, overall, collisions decrease the ability of AFL to discover bugs.

Table 10. Trimming experiment score

Fuzzer	**Average normalized score**
`AFL collisions free`	96.52
`AFL`	89.44

Taking `arrow_parquet-arrow-fuzz` as showcase for this experiment (Figure 9), we can see how the collision-free variant performs better, even if by a small small number of bugs.

The collision free variant is better than vanilla AFL on six benchmarks (only two with statistically significant results) and worse on two benchmarks (none statistically significant).

It is very interesting to note that the only two targets in which vanilla AFL performs better are large programs: `wireshark_fuzzshark_ip` and `ffmpeg_ffmpeg_demuxer_fuzzer`. In both cases, the number of edges is greater than the shared map size, thus causing collisions also in our variant. While these collisions are considerably less (as they are only the ones due to the overflow of the map, which also affects AFL), AFL uses a hash function and therefore its collisions are equally distributed across the map. Our variant uses instead sequential identifiers, and therefore all colliding blocks are located in the same area of code – basically eclipsing an entire area of the target from the fuzzer.

In AFL++, this problem is solved by increasing dynamically the map size, which however introduces a significant slowdown as the processing of the shared memory is the second most expensive operation in AFL.

> This experiment shows how the simpler approach that reduces collisions by using edges enumeration is marginally better. We think that the classic hash-based index is still used in AFL as a legacy indexing scheme borrowed from `afl-gcc` and never changed. Researchers that want to build new fuzzers upon AFL should therefore consider replacing such an indexing scheme.

5.10 Discussion

AFL includes multiple features whose impact has never been properly evaluated. Many of them, such as hitcounts, are commonly re-used by other fuzzers and taken for granted for historical reasons. Our work highlights the importance of benchmarking each aspect of a fuzzer in isolation to fully understand if, and when, it is beneficial for a testing campaign.

For instance, the results of our experiment show that researchers, designing new fuzzers, need to think twice before replacing certain features of AFL, such as novelty-search, with their own. On the other hand, some features, such as hitcounts, may not be beneficial for certain targets, and better options may be available. Our findings also suggest that usability-oriented features (such as using splicing as a stage, instead of as a mutator) should be carefully considered and, if they are decreasing the fuzzer effectiveness, should probably not be used by default in fuzzers.

Our results also show that AFL is a complex tool, and therefore it might not be the correct baseline to use when evaluating novel ideas. In reality simpler approaches may perform better due to a variety of reasons. Therefore, new fuzzers (even if based on AFL) should adopt such approaches as true baselines, since outperforming AFL can be relatively easy.

Finally, our experiments emphasize the difficulty of drawing general conclusions. Simply reporting the *'average case'* can cover up some exceptional performance for single test cases or runs. In fact, our effort to precisely benchmark different aspects of AFL encountered three main problems:

- **Randomness**. Despite the fact that FuzzBench repeats each experiment up to 20 times to mitigate the random nature of fuzzing, most of the results were not statistically significant. This affected some tests more than others, depending on the actual magnitude of the impact of a given feature. The worse case we observed was in the experiments on timeouts, where even though results seem to suggest that longer timeouts are beneficial for slow targets, *none* of the results were statistically significant. Thus, conclusive experiments to fine-tune a fuzzer might require very large numbers of runs.
- **Target-dependency**. Even when results were statistically significant, the conclusions were often target-specific. In other words, the *"common sense"* the community derives from specific targets can be misleading and difficult to generalize, and specific targets often highlight how a feature that is often beneficial can be largely outperformed by other configurations in a specific case (novelty search is a great example of this behavior).
- **Introspection.** A final takeaway is that looking only at bugs and code coverage is often insufficient to really understand the effect and impact of a given technique. More fine-grained ways to introspect the operation

of a fuzzer in benchmarking tools (such as by reporting the numer of timing-out inputs) can greatly help the community to perform more quantitative measurements.

6 CONCLUDING REMARKS

This paper dissects the popular fuzzing project American Fuzzy Lop. We studied its implementation, analyzed each individual component, and demonstrated how their details impact the overall functionality. We provide a case study evaluating features of AFL over 25 applications from the Fuzzbench dataset.

Our experiments suggest that future fuzzers and fuzzing experiments need to be familiar with several aspects of AFL that may significantly affect the effectiveness of a fuzzing campaign. We confirm the positive effects of some aspects of AFL, such as the novelty search algorithm in Sec. 5.2, but also the negative impact of others, such as its testcase scoring, in Sec. 5.4. AFL's prior decisions affect evaluations of new research based on AFL. Researchers who clone and extend AFL need to be aware that AFL's implementation details will impact their research and the outcome of their experiments.

We hope that our study provides useful information for researchers and practitioners who, in the future, will have to work on the previously unevaluated aspects of AFL.

ACKNOWLEDGMENTS

We would like to thank Michał Zalewski for his incredible contribution AFL and for answering our questions about this tool, the rest of the AFL++ team and community for being awesome, and the anonymous reviewers for their constructive feedback. A thank you to Slasti Mormanti too for his valuable insights. This project has been supported by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA875019C0003.

REFERENCES

[1] [n. d.]. CERT Basic Fuzzing Framework. https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework. [Online; accessed 1 Sep. 2022].

[2] [n. d.]. Funfuzz MozillaSecurity. https://github.com/MozillaSecurity/funfuzz. [Online; accessed 1 Sep. 2022].

[3] [n. d.]. Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz. [Online; accessed 1 Sep. 2022].

[4] 2016. Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Online; accessed 22 Dec. 2021].

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

[6] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (2018), 39 pages. https://doi.org/10.1145/3182657

[8] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

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

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[11] Marcel Böhme and Soumya Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering* 42, 4 (2016), 345–360. https://doi.org/10.1109/TSE.2015.2487274

[12] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[13] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. {MEUZZ}: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. 77–92.

[14] Jared D. DeMott and R. Enbody. 2007. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing *(Black Hat USA)*.

[15] M. Eddington. [n. d.]. Peach fuzzing platform. https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html. [Online; accessed 22 Dec. 2021].

[16] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3395363.3397372

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

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

[20] Patrice Godefroid. 2007. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. 1–1.

[21] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2016. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 517–528.

[22] Jesse Hertz and Tim Newsham. [n. d.]. Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/ [Online; accessed 1 Sep. 2022].

[23] Marc Heuse. 2020. afl-clang-lto - collision free instrumentation at link time. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md. [Online; accessed 1 Sep. 2022].

[24] 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* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[26] LLVM. [n. d.]. SanitizerCoverage - Edge coverage. https://clang.llvm.org/docs/SanitizerCoverage.html#edge-coverage. [Online; accessed 1 Sep. 2022].

[27] LLVM Project. 2018. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. [Online; accessed 1 Sep. 2022].

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

[29] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1024–1036. https://doi.org/10.1145/3377811.3380421

[30] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. 2020. *Fitness Guided Vulnerability Detection with Greybox Fuzzing*. Association for Computing Machinery, New York, NY, USA, 513–520. https://doi.org/10.1145/3387940.3391457

[31] 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.

[32] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[33] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[34] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360600

[35] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681

[36] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*.

[37] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 181–198.

[38] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[39] 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)*. USENIX Association, Vancouver, B.C. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[40] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 167–182.

[41] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2021. Nyx-Net: Network Fuzzing with Incremental Snapshots. *arXiv preprint arXiv:2111.03013* (2021).

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, 28.

[43] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[44] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing. (2021).

[45] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[46] Harmen-Hinrich Sthamer. 1995. *The automatic generation of software test data using genetic algorithms.* Ph. D. Dissertation. University of Glamorgan.

[47] Robert Swiecki. [n. d.]. Honggfuzz. https://github.com/google/honggfuzz. [Online; accessed 1 Sep. 2022].

[48] Fabian Toepfer and Dominik Maier. 2021. BSOD: Binary-only Scalable fuzzing Of device Drivers. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 48–61.

[49] Dmitry Vyukov. [n. d.]. syzkaller - kernel fuzzer. https://github.com/google/syzkaller [Online; accessed 1 Sep. 2022].

[50] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[51] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *NDSS*.

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

[53] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2313–2328. https://doi.org/10.1145/3133956.3134046

[54] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[55] Michał Zalewski. [n. d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. [Online; accessed 1 Sep. 2022].

[56] Michał Zalewski. 2014. Binary fuzzing strategies: what works, what doesn't. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html. [Online; accessed 1 Sep. 2022].

[57] Michał Zalewski. 2014. Fuzzing random programs without execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html. [Online; accessed 1 Sep. 2022].

[58] Michał Zalewski. 2015. afl-fuzz: making up grammar with a dictionary in hand. https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html. [Online; accessed 1 Sep. 2022].

[59] Michał Zalewski. 2016. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt. [Online; accessed 1 Sep. 2022].

[60] Michał Zalewski. 2016. Bunny the Fuzzer. https://code.google.com/archive/p/bunny-the-fuzzer/. [Online; accessed 1 Sep. 2022].

[61] Michał Zalewski. 2016. "FidgetyAFL" implemented in 2.31b. https://groups.google.com/g/afl-users/c/1PmKJC-EKZ0/m/zck6Iu77DgAJ. [Online; accessed 1 Sep. 2022].