Registered Report: Dissecting American Fuzzy Lop
A FuzzBench Evaluation

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Alessandro Mantovani
EURECOM
mantovan@eurecom.fr

Dominik Maier
TU Berlin
dmaier@sect.tu-berlin.de

Davide Balzarotti
EURECOM
balzarot@eurecom.fr

***Abstract*—AFL is one of the most used and extended fuzzing projects, 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 comparing different metrics used to evaluate fuzzers. This will prove the efficacy of some patterns and clarify which aspects are instead outdated. To achieve this, we set up 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 the scheduling methodologies.

Our preliminary 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 target coverage, other features are related to usability and reliability. Most important, the outcome of our experiments will indicate which parts of AFL we should preserve in modern fuzzers.

I. 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 [24], [27], OS kernels [42], [35] and virtual machine hypervisors [34].

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 how some novel aspects can be beneficial to unveil new vulnerable flaws. Although each new tool comes with a new set of features that distinguish it from the other variants, a considerable amount of the fuzzer functionalities is usually inherited from the "parent" project, which is often a well-established tool in the community.

This is a preregistered report for the 1st International Fuzzing Workshop (FUZZING '22) https://fuzzingworkshop.github.io/
April 24, 2022
More info about preregistration and experiments in fuzzing at http://fuzzbench.com/blog/2021/04/22/special-issue/

Over the past five years, both industrial and academic research on fuzz testing has reached a consensus into improving one single tool – the American Fuzzy Lop (AFL) [48] released in 2013 by Michał Zalewski. Two main aspects 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 uncovering vulnerabilities with a relatively low effort for the security analyst. 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.

Our paper provides an accurate analysis of many internal mechanisms, parameters, and design choices 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 actual consequences. In many cases, improvements came from contributions outside the academic ecosystem, thus lacking experiments and clear results to demonstrate why a specific design choice was needed or was better than other 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 can affect the results of a fuzzing experiment, both positively or 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 positive design choices must be 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, and should, 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 (e.g., the scheduling, the mutation, the feedback). We exclude instead 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. Therefore, in this case, we patched AFL to include an alternative approach to measure the coverage, namely *plain edge coverage*.

While we identified *nine* unique aspects to study, for our preliminary evaluations we focused on two features that are crucial in modern fuzzers: the coverage measurement and the criteria used to establish if a particular testcase is interesting or not. More specifically, for these two comparisons, we ran a set of experiments on the popular FuzzBench benchmarking service [27]. We plan to extend this evaluation to the remaining seven features in an extended version of this work.

Despite the restricted focus, our experiments already show interesting findings. For instance, the fact that *plain edge coverage* is capable in many cases to outperform *hitcounts*. This finding suggests that a fuzzing campaign should integrate both solutions to maximize the probabilities of success. We also found that relying on a fitness function alone does not positively impact the fuzzing session. In contrast, novelty search, one of the major and often forgotten contributions of AFL, remains to date the best alternative to evaluate whether a testcase is interesting or not.

These are only two examples of the results we aim to obtain with our methodology and experiments. We hope that our preliminary evaluation case study can pave the way for a more sound and complete set of experiments and therefore, in the spirit of open-science, we plan to release our code and artifacts upon publication.

II. Fuzz Testing

Fuzz testing, or *fuzzing*, is a popular vulnerability discovery technique that executes a target several times and, for each run, mutates the input to trigger novel and potentially buggy program points in the Software Under Test (SUT).

The first fuzzers appeared in the early '90 [28], primarily relying on some forms of *blackbox* testing. In this case, the fuzzer provided the SUT 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 [14], 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 if the mutated input results in a crash immediatly, 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 [17] relies on complex instrumentation and code analysis to produce more "interesting" inputs at the price of introducing a non-negligible performance slowdown [37]. On the other hand, methodologies that aim to reach the performances of blackbox fuzzers and to drive their exploration using lightweight code instrumentation fall under the category of greybox approaches. In this case, the code injected in the SUT 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 [36], undefined behaviors like integer overflows [4] and other more specific classes of bugs [18].

Despite the advances and improvements that made the new generation whitebox fuzzers much more performant [32], greybox approaches remain the leading technique to discover vulnerabilities in modern codebases. For instance, the OSS-Fuzz project by Google [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 perform the mutation of the testcase content to increase the chances to stress new behaviors. Traditional uninformed mutation strategies [49] only work properly for some types of input and some applications, as those that perform binary format parsing. More recently, approaches like AFLSmart and Zest [31], [29] suggested focusing the mutation on a higher-level structure rather than on the raw bytes, for instance, by introducing AST-like representations of the input. The community also introduced the concept of *grammar-awareness* to indicate a fuzzer's capabilitiy to mutate an input according to certain grammar rules [5], [38]. Always in the scope of testcases management, another line of research focused on testcase scheduling by trying to maximize the explored locations by optimizing the selection of the inputs present in the corpus [44], [12].

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 SUT, such as *block coverage*, that rewards the fuzzer when it hits a new basic block, and later, *edge coverage*, that instead focuses the attention on newly discovered edges inside the CFG. These mechanisms allow a fuzzer to keep only those testcases that result in new code coverage, leading the fuzzing campaign to go deeper in the code and eventually to reach

the buggy location. This simple idea is at the base of many modern fuzz testing projects, such as AFL [48], AFL++ [16] and libfuzzer [23], 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.

By extending from the concept of feedback-guided fuzzing, researchers have proposed new forms of feedback over a target execution, in the attempt to reveal different program locations or states not easily reachable by traditional techniques [11], [43]. Alternative forms of feedback may evaluate the quality of a testcase independent on code coverage, but according to different aspects of the execution [6], [30], [15], [25].

III. AMERICAN FUZZY LOP

American Fuzzy Lop is a mutational coverage-guided fuzzer with a suite of additional tools. These include testcase- and corpus-minimizers, a fault-triggering allocator, and a file format analyzer. It's lastest version is the 2.57b [1], released in 2020, but the fuzzer is unmaintaned by the 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.

A. General Design

As stated by Zalewski in a technical whitepaper [52] 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 the amount of coverage reached. While some choices in AFL improve these two metrics, the principle of ease of use is often forgotten, despite the fact that 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 with respect to 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 tasks 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 [54]. The testcase delivery to the target program is performed via standard input or via file. Finally, the target execution is controlled by using a *forkserver* [50], 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)`.

B. 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 [40], [13], 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*) that track the number of times an edge was executed. According to Zalewski [52], the use of hitcount buckets allows AFL to tackle path explosion problem.

Implementation-wise, AFL keeps a shared bitmap between the target and the fuzzer of 64kb (to match the L2 cache size at the time AFL was 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, introducing collisions in the bitmap. Starting from version 2.37b (released in 2017), AFL adopted the `trace-pc-guard` option of SanitizerCoverage [22] 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++ [16] and LIBFUZZER [23].

This coverage information is used in the fuzzer in different algorithms. The most important is the procedure used to decide if a testcase is *interesting*, and therefore whether it is worth adding it to the corpus. For this, AFL uses a *novelty search* algorithm that considers an input that uncovers a new entry in the map, never seen so far, or a value reaching a previously unseen bucket, as interesting.

The nature of the 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 was taken to avoid path explosion and to increase speed thanks to the optimized processing of the coverage map.

C. 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 is skipped with a 99% probability, otherwise the probability goes down to 95% in case of non-favored but fuzzed before entry and 75% for never selected cases.

[1] https://github.com/google/AFL/releases/tag/v2.57b

[2] https://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

Another scheduling application is the so-called *energy assignment* [9]. For each corpus entry, AFL calculates a score that is used to compute how many executions must be performed in each stage in which a 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 of faster than the global average from 0.1x up to 3x. The other metric is the number of filled entries in the coverage map when executing the testcase, in this case using 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 multiplier to fuzz for more time derived inputs that could have been difficult to discover by blackbox approaches.

D. Mutators

AFL relies on generic, target-agnostic, byte-level mutators [49]. 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 does not contribute to a change in coverage to avoid to mutating them in subsequents 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 is about the dictionary [51]. The dictionary is a set of tokens related to the input format, for instance `\x7fELF` if the target is an an ELF parser. The tokens can be user-specified (`-x` parameter in AFL) to help the fuzzer to generate testcases that are otherwise impossible to create using generic bit-level mutations or auto-detected by the fuzzer during the bit flips stage looking for groups of bits that, when changed, produce always the same coverage hinting that they are part of a magic value. The dictionary stages walk the input replacing and inserting each item in the both dictionaries.

The first non-deterministic mutation stage is *random havoc*. This applies several mutations such as 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 a full cycle of the queue without any new finding, but it is always enabled in FidgetyAFL [54]. 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 in the crashes folder keeps the information about the parent testcases, that can be up to 2, and about the mutations that were applied to them. This allows to reconstruct the entire process of derivation of a testcase, an information that an analyst can use during crash analysis.

E. Minimization

Some mutations can increase the size of a testcase and, especially for inputs discovered later in a testing campaign, can result in files with a very large size. These large, slow-to-parse inputs, can decrease AFL speed and 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. While this reduces the complexity of the items in the corpus, it also requires some additional executions for each testcase that is saved in the queue.

F. Instrumentation

To get the coverage information from each execution of the target, AFL employs several instrumentation options. First of all, it can instrument the compiler on x86 by intercepting the assembler and changing it to log each basic block using functions in an injected runtime in assembly. In addition, AFL also provides a LLVM pass [21] in which each block ID is assigned randomly at compile-time and the instrumentation to hash the blocks and write to the shared memory is inlined, 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 to fork when fuzzing stateless code, resulting in an increased performance.

Alongside compiler-based approaches, AFL comes with a binary-only mode: 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.

IV. 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 [54] as it is covered in the FuzzBench paper [27], which highlights that FidgetyAFL always outperforms AFL in terms of code coverage over time.

We propose to assess the contribution of each feature on the performance of AFL in terms of uncovered bugs and code coverage over 23h using FuzzBench [27]. When there is no such contribution, we will provide a qualitative explanation

of the possible impact of that feature on usability. In case of results that highly depends on the structure of the target program, we will try to classify manually wich kind of program is influenced by the tested feature.

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

1) Hitcounts: Hitcounts are adopted by other fuzzers today [23], [41], 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.

2) Novelty search vs. maximization of a fitness: While AFL considers every new discovered hitcount as interesting, both other early fuzzing solutions [53] and more recent tools [33] 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 [33].

We think that a big part of the success of AFL in terms of performance is the novelty search-based approach to evaluate interesting testcases. In order 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:

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

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 using a fitness are coupled with heavy static analysis or complex approaches using many features, not just code coverage [26]. 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 [45] 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 [33]. We expect the novelty search to outperforms 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) while a set of diverse testcase as the ones saved by AFL is better in a corpus.

3) Corpus culling: The prioritization of the small and fast testcases in the AFL corpus selection algorithm trades speed with the fuzzing of more complex testcases that often corresponds to complex program states. We want to benchmark this feature 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 [39].

In this experiment, we want to assess the difference in using the AFL corpus culling mechanism versus using the entire corpus. We expect faster growth in coverage over time and, potentially, more bugs triggered in the same time window. Preliminary experimts suggest, however, that the fuzzer without corpus culling may find bugs that AFL does not trigger.

4) 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 the focus on many derived works (e.g. [9], [47], [8]) as, therefore, it is an essential piece of AFL.

In this experiment, we want to measure the delta between the AFL solution and the baseline, represented by a constant and a random score. 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 will select a random number between this boundary. 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 in the AFL history. We expect that the major contribution is the prioritization of the novelties. So we foresee a small delta between the baselines and the patched AFL with the naive score calculation.

5) 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 the corpus structure as a queue for granted.

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

6) 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 of 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 [23]) often implement splicing as a mutation rather than a stage, thus applying it many more times for each testcase during their havoc stage.

[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.

We modified the AFL codebase to implement splicing as a mutation operator to compare the two. We hypothesize that this choice can have some consequences 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.

7) Trimming: Trimming the testcases allows the fuzzer to reduce the size of the input files and consequently give priority to small inputs, under the assumptions that large inputs introduce a slowdown in the execution and 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 shrunk testcase is stored.

Despite the fact that this algorithm can bring the two important benefits described above, we argue that reducing the size of the testcases could lead to lose state coverage. Additionally, the trimming phase could become a bottleneck for slow targets. Therefore, in our evaluation we plan to compare the default version of AFL against a modified one, where we disabled trimming. Our hypothesis is that trimming can be either beneficial or detrimental depending on the type of target program and the structure of its input.

8) Timeouts: The timeout regulates the maximum amount of time the target program runs for. This greatly influence the execution time of the target and in turns the number of executions per second. While the user can specify an arbitrary value by passing a command line option (`-t`), in the average case AFL can automatically compute a timeout value 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 try to modify the multiplicative factor to measure its affect on the fuzzing session. We expect that a higher timeout can lead to a better coverage, but also degrade the performance of the fuzzer. This trade-off might deserve a more careful study to properly tune this parameter depending on the target.

9) Collisions: As explained in section III-F the AFL approach to instrument the source code of the target programs consists of assigning an identifier for each basic block at compile-time. When using SanitizerCoverage [22]'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 [19].

In our evaluation, we want to compare the AFL instrumentations approach against a collision-free one. As SanitizerCoverage traces each block 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 incur into collisions during the fuzzing session as many implementations indeed make use of the `guard_var` 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 *pc-guard*, raising the question why random identifiers are 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 the classic approach too in order 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 improves a lot the speed of the fuzzer – as we want to evaluate the impact of the collisions in isolation.

V. Preliminary Evaluation

As a preliminary evaluation, in this section, we present the results of the first two sets of experiments, conducted by using the FuzzBench service [27]. We mainly use the bug benchmark of FuzzBench, which consists of 25 targets known to contain bugs, as we believe that uncovered bugs is the ultimate metric in fuzzing evaluation [20]. 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. The reported results are median values over 20 trials to mitigate the effects of randomness in fuzzing and the Mann-Whitney U test is used 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 using an average normalized score [27]. All the variants run using the `trace-pc-guard` instrumentation and persistent mode to mitigate the well-known impact [46] of `fork(2)`.

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.

Note that, for the purpose of the workshop, we carried out only the first two proposed experiments as a case study of the entire proposed evaluation.

A. Hitcounts

In this first set of experiments, we compare vanilla AFL against a modified version that does not use hitcounts. Table I reports the average normalized score of the number of uncovered bugs in our experiments [4]. Quite surprisingly, the AFL variant without hitcounts discovered more bugs than the unmodified AFL, a counter-intuitive result as hitcounts enable AFL to bypass coverage roadblocks that depend on loop counts.

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

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

TABLE I: Hitcounts vs. plain edge coverage bug-based experiment score

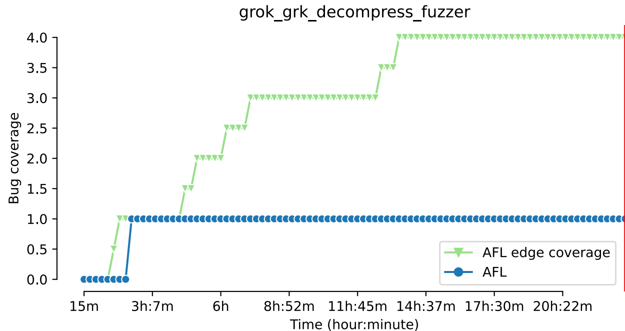

Fig. 1: Median bug coverage growth on grok (Hitcounts vs. plain edge coverage experiment)

In particular, AFL performed better on 6/25 benchmarks in terms of median discovered bugs, of which only two are statistically significant for the Mann-Whitney U test. The variant with only edge coverage is 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 clearly observe that the graphs reporting bugs uncovered over time (Fig. 1) and coverage over time (Fig. 2) 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 [43], [44], [15], 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 dependant, 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.

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

TABLE II: Hitcounts vs. plain edge coverage code coverage-based experiment score

To further confirm our intuition that hitcounts introduce a benefit only on some targets, we run another set of experiments

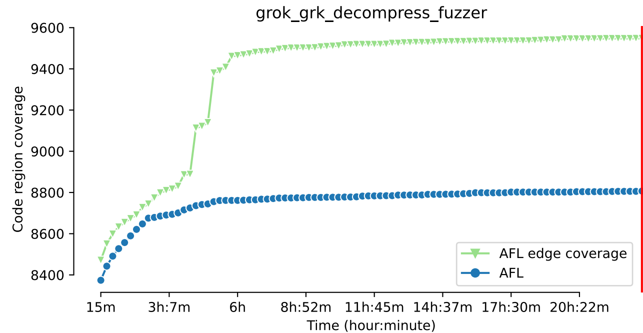

Fig. 2: Median code coverage growth on grok (Hitcounts vs. plain edge coverage experiment)

on FuzzBench on a different set of 22 benchmarks that FuzzBench uses to evaluate fuzzers using only code coverage as a metric[5]. The score reported in Table II 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 a highly variadic behavior suggests that users should include a variant without hitcounts when doing parallel or ensemble fuzzing like OSS-Fuzz [3].

B. *Novelty search vs. maximization of a fitness*

In this second experiment, we compare three fuzzers: vanilla AFL (that uses a novelty search), a variant with only fitness maximization, and a hybrid variant with both maximization and novelty search. We ran a bug-based benchmark[6] and found that, in average, vanilla AFL is the best performer – this time matching our expectation. In Table III we report the average normalized score of the number of uncovered bugs.

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

TABLE III: Novelty search vs. maximization of a fitness bug-based experiment score

The usage of the fitness only is clearly detrimental and the combination of both techniques does not introduce a

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

[6] https://www.fuzzbench.com/reports/experimental/2021-12-16-afl-fitness-bug/index.html

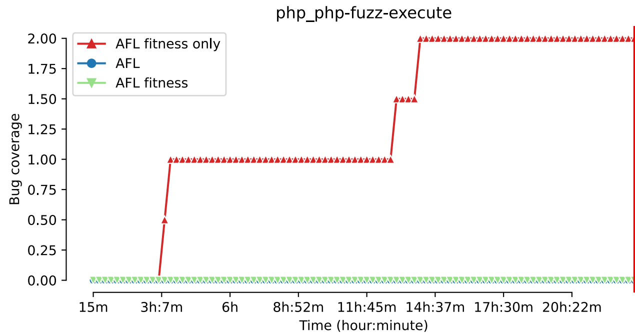

Fig. 3: Median bug coverage growth on PHP (Novelty search vs. maximization of a fitness)

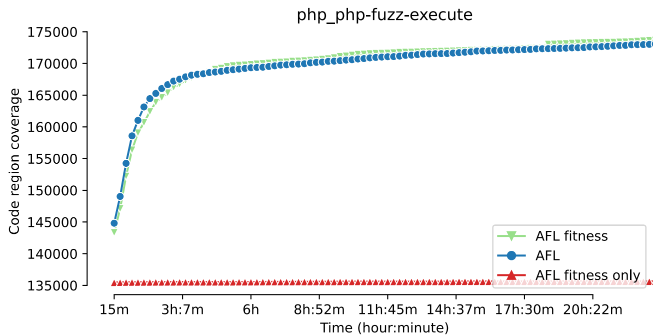

Fig. 4: Median code coverage growth on PHP (Novelty search vs. maximization of a fitness)

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 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 best fuzzer on 4/25 benchmarks, all statistically significant.

Unlike in the previous experiment, this time there is no correlation between uncovered code and bugs (Fig. 3 and 4). On this target, while the others two coverage-guided fuzzers start to explore the program (Fig. 4), a very large one with thousand of edges, the code coverage growth for the fitness only variant is negligible. The saved testcases in the corpus cover the same regions of the initial testcases so we can observe that, on this target, the fuzzer is behaving like a blackbox fuzzer without any coverage tracking capability.

This hints 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 [10] 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.

C. Discussion

Our preliminary studies showed that even core contributions of AFL, like hitcounts, have never been benchmarked fully and that the community still has much to learn about it. What is given as "common sense" in the fuzzing community about AFL can be true or false because observed only on specific targets, and, on the contrary, some specific targets can highlight how an almost always good feature can be outperformed in special cases, as we showed with the experiment about the novelty search.

We propose to the academic community and to practitioners an evaluation of the relevant features of AFL on a large dataset that is FuzzBench, in isolation, and with an understanding about what is working and what does not.

VI. 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 preliminary experiments suggest that future fuzzers and fuzzing experiments need to be aware of crucial aspects of AFL that affect every run significantly. We confirm the positive effects of AFL's novelty search algorithm in Sect. V-B. While our experiments in Sect. V-A prove the overall positive impact of hitcounts, we also show that they are target-dependent, and other forms of edge-coverage can yield better results in some instances. AFL's prior decisions affect evaluations of new research based on AFL. Researchers need to take the observation into account that AFL's implementation details will impact their new research ideas when they simply clone and patch AFL.

We hope that our study provides base knowledge for researchers and practitioners who, in the future, will have to work on the unevaluated aspects of AFL. This paper, and the future experiments we discuss, serve as an accessible resource and present a variety of insights about the internal, previously not yet sufficiently tested, design choices.

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] "CERT Basic Fuzzing Framework," https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework, [Online; accessed 20 Dec. 2021].

[2] "Funfuzz MozillaSecurity," https://github.com/MozillaSecurity/funfuzz, [Online; accessed 20 Dec. 2021].

[3] "Google OSS-Fuzz: continuous fuzzing of open source software," https://github.com/google/oss-fuzz, [Online; accessed 20 Dec. 2021].

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

[5] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," in *NDSS*, 2019.

[6] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "IJON: Exploring deep state spaces via fuzzing," in *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[7] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 50:1–50:39, 2018. [Online]. Available: http://doi.acm.org/10.1145/3182657

[8] M. Böhme, V. Manès, and S. K. Cha, "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*, ser. ESEC/FSE, 2020, pp. 1–11.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[10] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, 2016.

[11] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[12] Y. Chen, M. Ahmadi, B. Wang, L. Lu *et al.*, "{MEUZZ}: Smart seed scheduling for hybrid fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, 2020, pp. 77–92.

[13] J. D. DeMott and R. Enbody, "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing," ser. Black Hat USA, 2007.

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

[15] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2829–2846. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[16] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[17] P. Godefroid, "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)*, 2007, pp. 1–1.

[18] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "Typesan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 517–528.

[19] M. Heuse, "afl-clang-lto - collision free instrumentation at link time," https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md, 2020, [Online; accessed 20 Dec. 2021].

[20] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[21] C. Lattner and V. Adve, "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, Mar 2004.

[22] LLVM, "SanitizerCoverage - Edge coverage," https://clang.llvm.org/docs/SanitizerCoverage.html#edge-coverage, [Online; accessed 20 Dec. 2021].

[23] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, Sep. 2018, [Online; accessed 20 Dec. 2021].

[24] V. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, no. 01, oct 5555.

[25] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1024–1036. [Online]. Available: https://doi.org/10.1145/3377811.3380421

[26] R. K. Medicherla, R. Komondoor, and A. Roychoudhury, *Fitness Guided Vulnerability Detection with Greybox Fuzzing*. New York, NY, USA: Association for Computing Machinery, 2020, p. 513–520. [Online]. Available: https://doi.org/10.1145/3387940.3391457

[27] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "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*, 2021, pp. 1393–1403.

[28] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, Dec. 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[29] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 329–340. [Online]. Available: https://doi.org/10.1145/3293882.3330576

[30] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "FuzzFactory: Domain-specific fuzzing with waypoints," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360600

[31] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[32] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 181–198.

[33] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[34] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[35] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "KAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, pp. 167–182.

[36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USENIX Association, 2012, p. 28.

[37] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[38] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," 2021.

[39] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[40] H.-H. Sthamer, "The automatic generation of software test data using genetic algorithms," Ph.D. dissertation, University of Glamorgan, 1995.

[41] R. Swiecki, "Honggfuzz," https://github.com/google/honggfuzz, [Online; accessed 20 Dec. 2021].

[42] D. Vyukov, "syzkaller - kernel fuzzer," [Online; accessed 20 Dec. 2021]. [Online]. Available: https://github.com/google/syzkaller

[43] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "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)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[44] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," in *NDSS*, 2021.

[45] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *NDSS*, 2020.

[46] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2313–2328. [Online]. Available: https://doi.org/10.1145/3133956.3134046

[47] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "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, Aug. 2020, pp. 2307–2324. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[48] M. Zalewski, "American Fuzzy Lop," https://lcamtuf.coredump.cx/afl/, [Online; accessed 20 Dec. 2021].

[49] ——, "Binary fuzzing strategies: what works, what doesn't," https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html, 2014, [Online; accessed 20 Dec. 2021].

[50] ——, "Fuzzing random programs without execve()," https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2014, [Online; accessed 20 Dec. 2021].

[51] ——, "afl-fuzz: making up grammar with a dictionary in hand," https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html, 2015, [Online; accessed 20 Dec. 2021].

[52] ——, "American Fuzzy Lop - Whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016, [Online; accessed 20 Dec. 2021].

[53] ——, "Bunny the Fuzzer," https://code.google.com/archive/p/bunny-the-fuzzer/, 2016, [Online; accessed 20 Dec. 2021].

[54] ——, ""FidgetyAFL" implemented in 2.31b," https://groups.google.com/g/afl-users/c/1PmKJC-EKZ0/m/zck6Iu77DgAJ, 2016, [Online; accessed 20 Dec. 2021].