AFL++: Combining Incremental Steps of Fuzzing Research

Andrea Fioraldi[†], Dominik Maier[‡], Heiko Eißfeldt, Marc Heuse[§]
{andrea, dominik, heiko, marc}@aflplus.plus
†*Sapienza University of Rome*, ‡*TU Berlin*, [§]*The Hacker's Choice*

Abstract

In this paper, we present AFL++, a community-driven open-source tool that incorporates state-of-the-art fuzzing research, to make the research comparable, reproducible, combinable and — most importantly – useable. It offers a variety of novel features, for example its *Custom Mutator API*, able to extend the fuzzing process at many stages. With it, mutators for specific targets can also be written by experienced security testers. We hope for AFL++ to become a new baseline tool not only for current, but also for future research, as it allows to test new techniques quickly, and evaluate not only the effectiveness of the single technique versus the state-of-the-art, but also in combination with other techniques. The paper gives an evaluation of hand-picked fuzzing technologies — shining light on the fact that while each novel fuzzing method can increase performance in some targets — it decreases performance for other targets. This is an insight future fuzzing research should consider in their evaluations.

1 Introduction

The research on Fuzzing is a flourishing field. Fuzzing uncovers a variety of bugs in a fully automated fashion. Fuzz-testing has seen a big interest in the information security community in recent years and has sparked advancements in different fields. In tests performed by Shoshitaishvili et al., symbolically-assisted fuzzing identified almost three times more vulnerabilities than symbolic execution [39].

The number of developed techniques aiming to improve fuzzing grows [28] — sometimes without fully-functioning code, if at all. In addition, fuzzing techniques are often developed orthogonally and independently, so combining them can be a long process. It can be difficult for industry and the OSS community to decide which research is worth the attention. Instead, they may stick with a basic setup, even though modern research would find more bugs for their target, faster. On the other hand, researchers themselves can have a hard time evaluating their novel tools, and may find themselves unable to combine functionality with the compatible techniques that address different, but related problems in fuzzing — for example picking a recent seed scheduling for their mutator. A new feedback concept may not live up to its full potential if it cannot be combined with existing techniques solving other problems — like overcoming hard comparison instructions — reducing the impact of the research on paper due to lackluster statistics.

In this paper, we try to solve these problems by raising the bar of broadly available, research-backed, fuzzing, and by giving researchers an extensible API to build upon. We propose a novel fuzzing framework, AFL++. Future research can use AFL++ as a new baseline. It gives researchers the possibility to evaluate combinations of their proposals with state-of-the-art orthogonal features already implemented in AFL++ — with a highly reduced implementation effort. At the same time, it offers industry professionals a large range of easy-to-use features adapted from cutting-edge research, that can greatly improve the outcome of a fuzzing campaign. AFL++ is a reengineered fork of the popular coverage-guided fuzzer AFL by Zalewski [47] which has proven to be a solid base for works in academia and industry alike.

AFL was chosen as the base because at the time of the start of this project it was already unmaintained for 18 months while at the same time a lot of community patches and academic forks were available. Hence this provided a perfect start. This would not have been possible with LIBFUZZER and HONGGFUZZ as they were and still are actively maintained and still do not enjoy major forks and enhancements in comparison (with notable exceptions for ENTROPIC [9] and Vranken's enhancements [41]).

While AFL++ started off as collection of patches and forks to AFL, over time we reimplemented non-afl-based research like REDQUEEN [5], as well as research-grade extensions to AFL to make them production ready, for AFL++. We then added novel features on top of this state-of-the-art, that will also be discussed in this paper.

All in all, this paper will give insights into a year of active, open-source, fuzzing research, discuss lessons-learned, and discuss the novel *Custom Mutator API*, a way to implement novel fuzzing research.

1.1 Contributions

1. We propose AFL++, incorporating recent fuzzing research into one usable tool.

2. We discuss AFL++'s novel *Custom Mutator API* an approachable, and future-proof way to implement and combine future research.

3. Using AFL++, we evaluate a selection of incorporated technologies and features against, and with, each other. We show how target-dependent each technology is — a very relevant insight for future research.

AFL++ and all the related artifacts are Open Source Software and available on GitHub at:

https://github.com/AFLplusplus

The test cases for this research are available at FuzzBench [22].

2 State-of-the-Art

American Fuzzy Lop (AFL) [47] is one of the most widely used and most successful coverage-guided fuzzers of all time. It is the current baseline for a wide variety of fuzzing-related publications. In this section, we discuss American Fuzzy Lop and the research done over the past years to improve specific aspects of this fuzzer in-depth, yet as concise as possible. The concepts explained in this section are directly relevant for AFL++, which will be presented in Sect. 3.

2.1 American Fuzzy Lop

AFL is a mutational, coverage guided fuzzer. It mutates a set of test cases to reach previously unexplored points in the program. When this happens, the test case triggering new coverage is saved as part of the test case queue.

2.1.1 Coverage Guided Feedback

The coverage feedback of AFL is a hybrid metric, combining edge coverage with the count of how many times the respective edge was executed in one run. This count is bucketed to a power of two to avoid path explosion. An input is considered interesting (i.e. saved to the queue) if it explores at least one new bucket for an edge. These buckets, or *hitcounts*, are logged to a shared bitmap during execution, in which each byte represents an edge. The size of this map is limited, so collisions are possible. AFL employs an approximation of *weighted minimum set cover* to maintain a set of *favored* test cases — in terms of coverage — with speed and size as weights.

Using the coverage feedback AFL also tries, for each test case in the queue, to reduce the size of the test case and improve the speed of the target while maintaining intact the coverage in a stage called *trimming*.

2.1.2 Mutations

The mutations of AFL are divided into two categories: *deterministic* and *havoc*. Deterministic stages include single deterministic mutations on contents of the test cases, like bit flips, additions, substitution with integers from a set of common interesting values (e.g -1, INT_MAX, ...), and others. In havoc, mutations are randomly stacked and include also changes to the size of the test case (e.g. adds or deletes portions of the input). Additionally, at a later stage, AFL may merge two test cases into one and apply havoc, in the so-called *splicing* stage.

2.1.3 Forkserver

To avoid the overhead of `execve()`, AFL uses the so-called *forkserver*. The fuzzer injects a forkserver, controlled through an IPC mechanism, into the target. Whenever AFL needs to execute a test case, it writes the input, then tells the target to fork itself. The child will execute the test case, the parent process waits for this time. The forkserver can also fork later in the target. In this case, the fuzzer does not pay the cost of running the expensive initialization and startup routines each time.

2.1.4 Persistent Mode

Persistent mode greatly improves performance. Because `fork()` is known to be a bottleneck, for the Persistent Mode, the target does not fork for each test case. Instead, a loop can be patched into the target, executing one test case per iteration. To work, each iteration needs to leave cause minimal state changes.

2.2 Smart Scheduling

A modern coverage-guided fuzzer may implement different prioritization algorithms to schedule various elements in the fuzzing pipeline. The goal of schedulers is usually to improve overall coverage and bug detections through smart test case selection.

2.2.1 AFLFast

AFLFast [11] by Böhme et al. shows the need to stress low-frequency paths to explore more branches and find more bugs. They developed several improvements to AFL to not only stress common paths, with the goal to expose additional program behavior. They highlight two problems:

1. In which order should the fuzzer pick the seeds, in order to stress low-frequency paths?
2. Can we tune the amount of generated inputs from each seed (the energy)?

The authors address the first issue with a set of novel search strategies and the second by introducing six power schedules to compute the energy from parameters collected during the fuzzing process.

2.2.2 MOpt

As a horizontal problem to seed scheduling, MOPT [25] introduced mutation scheduling. In the work, Lyu et al., explore the possibility to give different probabilities to the mutation operators, using a custom *Particle Swarm Optimization* algorithm. This optimization improves the capabilities of a fuzzer to discover coverage quickly. In their patch to AFL, the authors divide the fuzzing stages into the following two modules. *Pilot*, a module that evaluates the operators, assigning probabilities based on the effectiveness. The *Code* module generates mutations, taking the probabilities found during Pilot into account.

2.3 Bypassing Roadblocks

Traditionally, coverage guided fuzzers suffer from roadblocks that prevent to explore code behind them. Typical roadblocks are larger comparisons, like a string, and checksum checks. A range of research was derived to tackle this problem.

2.3.1 LAF-Intel

LAF-INTEL [2] is a work that aims to bypass hard multi-byte comparisons, by splitting them into multiple single-byte comparisons. That way, these comparisons can be passed, byte by byte, with the coverage guided fuzzer receiving feedback for each part. The original implementation is a set of LLVM passes, splitting up integer comparisons, but also calls to string comparisons functions like `strcmp` when one of the arguments is known at compile time. In the details, LAF-Intel:

1. Simplifies the >= (and <=) operators into chains of > (<) and == comparisons;
2. Changes signed integer comparisons to a chain of sign-only comparison and unsigned comparisons;
3. Splits all unsigned integer comparisons with bit widths of 64, 32 or 16 bits to chains of multiple comparisons of 8 bit;

2.3.2 RedQueen

Recently, REDQUEEN [5], based on KAFL [36], explored the possibility to bypass hard comparisons and checksum checks, like other previous works in literature [35] [12] [33] [44], but without the use of expensive techniques like *taint tracking* [46] or *symbolic execution* [6, 37]. This fuzzer focuses on the comparisons that are defined as Input-To-State (I2S), a type of comparison that has a direct dependency with the input in at least one of its operands. The authors showed that many of the roadblocks comparisons are of this type and developed a technique to locate and bypass them. REDQUEEN firstly increases the entropy in the input in its *colorization* stage, replacing bytes with random data while maintaining the coverage of the test case. In this way, observing an operand of an I2S comparison, the fuzzer can reduce the number of guesses to locate its position in the input. REDQUEEN then mutates the input replacing the I2S tokens extracted from comparisons and use again this information to locate checksum checks and patch them out. At the end of each fuzzing stage, REDQUEEN use again I2S replacement to repair checksums of the newly generated interesting inputs. If it fails, the patched checksum is detected as a false positive the patch removed.

2.4 Mutate Structured Inputs

A common issue for fuzzers is that they may generate mostly invalid inputs, making the state of the program after the parsing stages inaccessible. A solution to this is the usage of an input model, effectively reducing the space of generated inputs. This allows a feedback-based fuzzer to explore deep paths in a program.

2.4.1 AFLSmart

Pham et al. introduced structured fuzzing to AFL: AFLSMART [34]. AFLSMART uses PEACH [14] pits as input model format, a widely used specification for structured black-box fuzzing. This choice makes it possible to re-use specifications for protocols written for PEACH. AFLSMART parses a test case the first time that it is extracted from the queue. It does so in a lazy way, with *deferred cracking*, that allows AFLSMART to fallback to AFL if it is good enough at exploring coverage without wasting time with parsing. The result of the parsing step is a virtual structure that represents an AST. AFLSMART introduces *higher-order* structural mutations, mutating the virtual structure instead of raw bytes. It can be configured to use only these structural mutations or stack them alongside the others in Havoc.

3 A New Baseline for Fuzzing

In this section, we will explain the engineering background of AFL++. The core of AFL++ is a forked version of AFL, a fuzzer on which a part of the academical fuzzing research is based upon, and which is also used extensively in the industry. This section describes what AFL++ adds on top, including many of the features discussed in Sect. 2. AFL++ is not limited to the features discussed here. The amount of smaller advancements in usability and engineering go beyond the scope of this paper. For a deep dive in this small but effective improvements, refer to the AFL++ documentation [18].

3.1 Seed Scheduling

AFL++ incorporates AFLFAST and extends it with additional power schedules. This included all schedules from AFLFAST: *fast*, *coe*, *explore*, *quad*, *lin*, *exploit*. These schedules are functions of the following variables:

1. The times that seed is chosen from the queue;
2. The number of generated inputs with the same coverage of the seed;
3. The average number of generated test cases with the same coverage in general;

The default schedule is *explore*. In addition to this, AFL++ adds the *mmopt* and the *rare* schedules. Mmopt increases the score for the newest seeds to help delving deeper into newly discovered paths. *Rare* ignores the runtime of the seed — unlike all other schedules — and additionally puts a focus on seeds with edges that are rarely covered by other seeds, an effective metric as shown in [24] [10].

3.2 Mutators

AFL++ incorporates more mutators than the traditional Deterministic and Havoc pipeline of AFL. The mutators can be used in combination with others.

3.2.1 Custom Mutator API

AFL++ can be easily extended for new research in academia and be adapted to specific targets for vulnerability discovery. For this, it offers an ever-growing API. The current state is as follows.

Custom mutators allow fuzzing research to build novel scheduling, mutation, and minimizations on top of AFL++, without forking and patching AFL, as is the case with a lot of current tooling. Initial support to this was first independently developed in the Holler's AFL fork [19], but got extended with a lot of new functionality. Plugins can be written in C ABI compatible languages, and even prototyped in Python. With the current API, for instance, AFLSMART can be rewritten completely as an AFL++ plugin. The following functions can currently be implemented:

`afl_custom_(de)init` Each custom mutator can use these self-explaining functions to initialize or deinitialize the module, `afl_custom_init` and `afl_custom_deinit`. The AFL++'s pseudo-random generator seed is passed to *init*. The custom mutator should then make sure that fuzzing results are reproducible given the same seed.

`afl_custom_queue_get` is a callback that determines whether the custom fuzzer should fuzz the current queue entry or not. In this routine, the user can also perform the initialization of associated metadata for an input, for instance, the virtual structure for structured fuzzing.

`afl_custom_fuzz` performs custom mutations on a given input. It accepts an additional test case.

`afl_custom_havoc_mutation` performs a single custom mutation on a given input. This mutation is stacked with the other mutations in the havoc stage. The `afl_custom_havoc_mutation_probability` returns the probability that the custom mutation is called in havoc enabling tuning (defaults to 6%, inspired by AFLSMART).

`afl_custom_post_process` In some cases, the format of the mutated data returned from the custom mutator is not suitable to directly execute the target with this input. For example, when using *libprotobuf-mutator*, the data returned is in a protobuf format which corresponds to a given grammar, that first need to be converted to the target's plain-text format. In such scenarios, or to fix checksums and sizes, the user can define the `afl_custom_post_process` function.

`afl_custom_queue_new_entry` is called after adding a new test case to the queue, a useful hook to store metadata on disk.

Trimming Support The generic trimming routines implemented in AFL++ (Sec. 2.1.1) may destroy the structure of complex formats. This is especially the case when your target can process a part of the input (causing coverage) and then errors out on the remaining input. In such cases, it makes sense to implement a custom trimming routine. The API consists of multiple methods because after each trimming step, the coverage bitmap has to be against the map before trim.

`afl_custom_init_trim` is called at the start of each trimming operation and receives the initial buffer. It should return the number of iteration steps possible on this input (e.g. if the input has n elements of which one should be removed, returning n-1). If the implemented trimming algorithm doesn't allow for determining the amount of (remaining) steps, then it can return 1 to indicate that further trimming could be performed, which will be performed while `afl_custom_post_trim` returns 0.

`afl_custom_trim` is called for each trimming operation. It memorizes the current state and hence can save reparsing steps for each iteration. It should return the trimmed input buffer, where the returned data must not exceed the initial input data in length.

`afl_custom_post_trim` is called after each trim operation to inform if the trimming step was successful or not (in terms of the same coverage). This method must return the next trim iteration index (from 0 to the maximum amount of steps returned in `afl_custom_init_trim`).

3.2.2 Input-To-State Mutator

AFL++ implements a mutator based on REDQUEEN's *Input-To-State (I2S)* replacement. In addition to what is described above, we made a few optimizations to improve upon the original implementation.

Firstly, colorization seems to be very effective in increasing the entropy of the bytes in the input but can slow down the fuzzer a lot, if, for instance, a critical field, such as a size field, is randomly mutated. We extended colorization to keep the mutated regions not simply when the hash of the coverage bitmap remains the same, but also when the execution speed remains in the bounds of a 2x slowdown from the original. This improvement seems to make the difference in pathological targets for REDQUEEN.

Another extension is a probabilistic fuzzing of each comparison. If the fuzzer fails to generate an interesting input when trying to bypass a comparison, the next time this comparison will be fuzzed with a lower probability. This avoids spending too much time on unsolvable comparisons that seem I2S but that are not.

CmpLog Instrumentation This mutator does not log comparisons operands using breakpoints, like in the original REDQUEEN implementation, but uses a shared table similar to the one used by Fioraldi et al. for WEIZZ [15]. Each comparison logs the operands of its last 256 executions in a 256 MB table shared between fuzzer and target.

The first part of the table maintains metadata for each comparison, like the size, the ID, and the real number of executions. The total size of 512 KB can be traversed in an efficient way in terms of cache locality. The metadata suffices to register if a comparison is not used, and the memory corresponding to the operand is never accessed. This instrumentation is available for the LLVM and QEMU instrumentations.

3.2.3 MOpt Mutator

AFL++ implements the *Core* and the *Pilot* mode of MOPT. In addition to this, MOPT was patched for AFL++, so that it can be combined with the Input-To-State mutator. On top, AFL++ supports interleaving of MOPT with the standard mutation modes.

3.3 Instrumentations

AFL++ supports several backends for instrumentation: *LLVM*, *GCC*, *QEMU*, *Unicorn* and *QBDI*. On top, it provides a proxy module that can be adapted to forward test cases to targets and give any kind of coverage to `afl-fuzz`, even remote and non-coverage, such as ampere consumption or branch addresses of JTAG.

Table 1 resumes the state of the implementation of the most important features discussed in Section 3 for each instrumentation backend.

NeverZero Orthogonally to the backend used for instrumentation, we developed an optimization to the hitcount mechanism of AFL. One problem of using a byte for the bitmap entries is, that the count of the edge executions can overflow. When this happens, we observed that if an edge is hit in multiples of 256 — overflowing the corresponding bitmap entry to 0 — the fuzzer is in an inconsistent state. We tried to solve this problem with two solutions, *NeverZero*, and *Saturated Counters*. The first avoids the overflow to 0 always adding the carry flag to the bitmap entry and so, if an edge is executed at least one time, the entry is never 0. The second freezes the counter when it reaches the value of 255. In a range of experiments, we observed that NeverZero is very effective and improves AFL in terms of coverage and speed (the seed selection now takes into account edges that were hidden before). Saturated Counters, however, decreases AFL's overall performance. We opted to make NeverZero the default for AFL++ on most of the available instrumentations. Saturated Counters are still available in a branch of the AFL++ repository for further research or reproduction.

3.3.1 LLVM

We support from LLVM [23] 3.4 up to LLVM 11, which is in beta at the time of writing. In LLVM mode, AFL++ supports a range of coverage metrics in addition to edge coverage [43]:

Context-sensitive Edge Coverage edge coverage is XORing the assigned ID of each block with the unique ID of the callee. This solution was firstly explored in [12] and seems effective in terms of code coverage, with the penalty to have more collisions and less speed.

Ngram instead of considering the previous block and the destination block when logging an edge, the fuzzer considers the destination block and the N-1 previous blocks where N is a number between 2 and 16.

In addition to the LLVM pass that instruments for coverage feedback, AFL++ ships several additional passes. All LAF-INTEL passes are included, with an experimental mode to split also floating-point comparisons that are abundant in software like video decoders or Javascript interpreters. Additionally the string comparison function analysis was improved to be able to process global and local variables when assigned a fixed string. The CmpLog passes are available, too, as discussed in the previous sections. A widely used feature, for example by Fratrik [29], is the list of files to instrument, firstly introduced by [19]. In LLVM mode the user can specify specific source modules to instruments. This is very useful, for instance, with

Table with supported features for each instrumentation backend

	afl-gcc	LLVM mode	GCC plugin	QEMU mode	Unicorn mode	QBDI mode
NeverZero	✓	✓		✓	✓	
Persistent mode		✓	✓	✓	✓	✓
LAF-Intel/ CompCov		✓		✓	✓	
CmpLog		✓		✓		
Instrument filelist		✓	✓	partial		
InsTrim		✓				
Ngram/Ctx coverage		✓				
Snapshot LKM		✓				

targets that process many input formats and the user wants to focus only on one of them. In persistent mode, in addition to the standard ways to pass the input to the target in AFL (stdin or file), AFL++ can also pass in new test cases through shared memory. This configuration brings an additional speedup of 2x to persistent mode, resulting in an overall speedup of up to 10-20x in respect to fork mode, in our initial tests.

AFL++ LLVM mode also implements the InsTrim [20] patches, combined with all the previously exposed features. InsTrim is an efficient way to select basic blocks when instrumenting in LLVM. It avoids to place useless instrumentation thanks to an analysis based on Dominator Tree. It reduces the number of instrumented locations to at least half of the standard instrumentation on most targets and so improves the fuzzer in terms of performance in speed.

3.3.2 GCC

Alongside the old `afl-gcc` wrapper, AFL++ ships a GCC plugin. It includes support to deferred initialization and persistent mode, like AFL LLVM mode. The supported features are not on par with LLVM, but additional features are planned in AFL++, with the goal of reaching feature parity eventually.

3.3.3 QEMU

The AFL QEMU patches for version 2.1 for binary-only fuzzing are almost completely replaced in AFL++ with a better set of patches based on QEMU 3.1.1. In comparison to other binary-only instrumentations, such as retrowrite based on binary patching [13], QEMU mode adds instrumentation at emulation time. The basic blocks transitions are now not anymore logged in the context of the emulator when selecting a block in QEMU but the call to the logging routine is inlined using a helper. In this way, we can re-enable the blocks linking that AFL disabled (as firstly shown in a thread-unsafe way by [8]) with an average speedup of 2-3x. The use of a helper enables also the use of Thread Local Storage, a concept not supported in TCG [3]. Recently, our QEMU mode was extended by Fioraldi, with QAsan [16], to support sanitization against heap violations incorporating a Dynamic Binary Translation based implementation of AddressSanitizer [38].

CompareCoverage In order to decrease the gap of features between source-level and binary-level fuzzing, AFL++ QEMU mode can split comparisons in a similar way to LAF-Intel using CompareCoverage [31]. Unlike the LLVM pass, the code is not modified, but all comparisons are hooked and each byte of each operand is compared, increasing a different bitmap entry if equal. This instrumentation is similar to the `popcnt` based instrumentation of LibFuzzer, but at a byte level, produceing so fewer inputs to avoid path explosion, an issue that makes the value-profile mode of LibFuzzer less effective than normal mode on some targets. It can be configured to split only integer comparisons with immediate operands, all integer comparisons, or all integer and floating-point comparisons.

Persistent Mode Unlike the old QEMU mode, AFL++'s QEMU-mode supports persistent mode. There are two main ways to achieve this:

1. Looping around a function: like WinAFL [17], the user can specify the address of a function, and automatically the fuzzer will use it in a persistent loop patching the return address. The address can be also not the first instruction of a function, but in this configuration, the user has to provide the offset on the stack to correctly locate the return address to patch;

2. Specify entry and exit points: a user can specify the address of the first and the last instruction of the loop and QEMU will emit code a runtime to generate a loop between those addresses;

This mode can even reach a 10x speedup and is recommended when possible.

3.3.4 Unicornafl

For fuzzing blob binaries like firmware, AFL++ incorporates a fork of *afl-unicorn* by Voss [40], which adds AFL support to the Unicorn Engine [30], called *unicornafl*. While the original version by Voss started the forkserver on the initial basic block, and was only available through the garbage-collected python, AFL++'s unicornafl adds a low-level C API, Rust and Python bindings, to interact with AFL++ directly. Unicorn already includes APIs to set page mappings, read and write memory and registers, add hooks, as well as start and stop execution with different conditions. AFL++-specific APIs allow the harness to kick off the fast persistent mode at any time, as well as to set multiple exits and post-fuzzing handlers to detect crashes.

The API offers `uc_afl_forkserver_start`, a specific call to kick off the fork server at a certain point in time, effectively freezing the current state prior to a fuzzing run and telling AFL++ to start generating inputs.

A special `uc_afl_fuzz` function serves as a one-stop-shop, directly reading input for each test case - with support for persistent mode. The target firmware is kept in the same state in the parent process, each fuzz test case is executed against a forked copy of the emulator. Furthermore, the forkserver contains a caching mechanism for Unicorn's JIT, inspired by the AFL QEMU mode. Unicornafl patches instrumentation into the translated blocks directly, reducing the need for indirect jumps, re-enabling the optimized blocks linking like already discussed for QEMU mode. The function `uc_afl_fuzz`:

1. Loads the current input.
2. Calls the `place_input_callback`. Here, the harness should write the input into the emulator memory at the appropriate position. For persistent mode, the emulator has to reset additional state changes in this step.
3. Emulates until one of the exits is reached, execution is cancelled by a hook, or an illegal state occurs.
4. Checks the Unicorn return and (optionally) calls the `crash_validation_callback`, where additional post-processing can be done to spot crashes.
5. For persistent mode, loops back to step 3.

The fuzz function also takes a list of exits at which emulation will stop, a flag whether the validation callback should also be called without a Unicorn error condition and an additional integer counter, indicating if—and how often—persistent mode should loop before forking again. Maier et al. were able to use it, on top of AFL++, to fuzz kernels [26] and even a cellular baseband rtos [27].

3.3.5 QBDI

AFL++ can fuzz Android libraries with compiler instrumentation using LLVM, but can also instrument closed-source libraries. It supports harnessing with QuarksLab's QBDI Dynamic Binary Instrumentation [21] framework for Android native libraries. An example is shipped with the source distribution of AFL++, making the usage and the extension very simple.

3.4 Platform Support

The support to several Operating Systems and distribution is maintained in AFL++. Besides GNU/Linux, the fuzzer runs on Android, iOS, macOS, FreeBSD, OpenBSD, NetBSD and is packaged in several popular distributions like Debian, Ubuntu, NixOS, Arch Linux, FreeBSD, Kali Linux and more. For this broad range of support, many features, like *libdislocator*, the AFL allocator to catch memory errors, had to be ported to several different Operating Systems and extended with previously unsupported allocation routines, like `posix_memalign()`. On top, AFL++'s QEMU [7] mode, has a Wine [1] mode, that can fuzz compatible Win32 binaries on GNU/Linux.

3.5 Snapshot LKM

The AFL state-restore mechanism based on `fork()` is well-known to be a performance bottleneck for a large number of targets. Hence, AFL++ integrates a Linux Kernel Module [1], inspired by Perffuzz by Xu [45]. Perffuzz implements a lightweight mechanism for process snapshot and restore. The average gain in performance on a single core with our module, compared to fork, is up to 2x, but the difference increases when running parallel fuzzing on many cores due to the locks in the kernel implementation of `fork()`. The use of snapshots, instead of fork, does not require a recompilation of the target program. Instead, once the driver is loaded, the module's presence is automatically detected.

4 Evaluation Use Cases

This section gives an insight into the fuzzing performance of some of the technologies discussed in Sect. 3 that are published research, and a usable part of AFL++. To show AFL++ in action, we also discuss combinations that were not possible otherwise, in their respective forked tools, such as the powerful combination of RedQueen and MOpt.

For the evaluation in this section, we use AFL++ together with FuzzBench [22] to reproduce and evaluate the state of the art (Evaluation of FuzzBench). AFL++ can be used to compare a wide range of fuzzing concepts against each other. In the following, we highlight a few interesting insights gathered over the last months in various tests. These are example runs, combinations, and evaluations that a new tool using AFL++ as baseline can do, with the help of FuzzBench.

As the range of features incorporated into AFL++ would exceed the scope of this paper, we picked 6 specific configurations:

1. **[Default]** the default AFL setup, with some specific fixes and improvements
2. **MOpt** A very effective mutator, discussed in Sect. 3.2.3.

[1] https://github.com/AFLplusplus/AFL-Snapshot-LKM

3. **Ngram4** A different instrumentation, interpreting the 4 following basic blocks as a unique path.

4. **RedQueen** Cmplog/RedQueen, an additional feedback channel to the fuzzer to reach greather depth, see Sect. 3.2.2

5. **Ngram4, Rare** The Ngram4 instrumentation, paired with Rare scheduling 3.1, a unique combination in AFL++.

6. **MOpt, RedQueen** The MOpt mutation, paired with RedQueen – another unique combination in AFL++.

The final data, discussed here, was collected using the FuzzBench service [22]. FuzzBench is a novel service by Google, offering fuzzing evaluations on a fixed set of 21 targets for originally 24 and now 23 hours each to all interested projects. Each run is redone about 20 times to get to a meaningful median of the Edge Coverage, as randomness in fuzz tests can produce accidental strong single runs, yet a good fuzzer should produce good results consistently.

From the total of 21 FuzzBench targets, we selected the following 9 due to their specific characteristics, showing visible outlier behavior. We selected 6 stand-out combinations we will discuss as part of this evaluation, from the larger number of AFL++ features we tested and looked at during the evaluation. All other test cases and targets not discussed here are available on FuzzBench for the avid reader in AFL++ specific and general FuzzBench runs. The final data for this evaluation was taken from FuzzBench run `2020-04-21-and-20-aflplusplus`[2].

With the selected 6 combinations, we try to show that:

- If we consider MOPT and Ngram4 as an example of possible novel techniques, we can get useful insights combining them with other orthogonal techniques like other mutators (REDQUEEN) and power schedules (rare);

- During this run, it becomes apparent, that all fuzzing behavior is highly target-specific so the importance of a good choice of a suitable configuration is crucial;

RedQueen In many areas, RedQueen is able to pass roadblocks no other configuration can, however not all targets have instructions where this method can help. One of these is the *libpcap* example, see Fig. 1g. Here, only RedQueen is able to reach any sorts of depth, as the I2S replacement allowed this configuration to bypass roadblocks the other configurations are unable to randomly guess. The MOpt mutator helps to further increase the coverage in this case. For *OpenThread* RedQueen also performs extremely well, fast, see Fig. 1a, but interestingly only if not paired with MOpt.

MOpt *MOpt* shows to either be a hit or miss - often it is highly effective, however when it is not, the performance is often very bad, with little middle ground. For *mbedtls*, see Fig. 1h, MOpt suddenly starts gaining a massive amount of new coverage. Mopt is able to find new paths in the middle of the run and outperform any other configuration by far. As this happens for multiple runs in the median, this is not a one-time event, but appears to be target-specific.

The combination RedQueen and MOpt in the *harfbuzz* target is the clear winner, see Fig. 1e, just as in the *libpcap* target, Fig. 1g, where only RedQueen reaches any sort of depth, and MOpt adds to the positive results. In the *bloaty fuzz target*, where RedQueen takes some time to catch up, MOpt proves very helpful, see Fig. 1b

For *libjpeg-turbo* MOpt has a high impact on the behavior of RedQueen, changing the median coverage graph almost completely, see Fig. 1d.

However, one very interesting observation can be made for the *lcms* target, where positive effects of RedQueen counters negative effects of MOpt in *lcms*, see Fig. 1i.

Ngram *Ngram* (of length 4 in our tests) can be an effective instrumentation method to reach more depth, as branch instrumentation depends on prior branches, effectively adding a bit of state feedback. The *zlib* example is such a case, see Fig. 1i. As is the case with almost all combinations, *Rare* scheduling sometimes improves, sometimes fares worse than the default AFL++ scheduling. Another target is *libxml2*, where MOpt performs badly, RedQueen has close to no effect, but Ngram4 shines 1c.

4.1 AFL++ Optimal

In the previous section we assessed the effectiveness of a range of fuzzing technologies implemented in AFL++ and how they interact when combined.

Typically, in real-word fuzzing campaigns, this is just the first step, and after a first evaluation of different configurations is carried out by a security researcher, the fine-tuning starts.

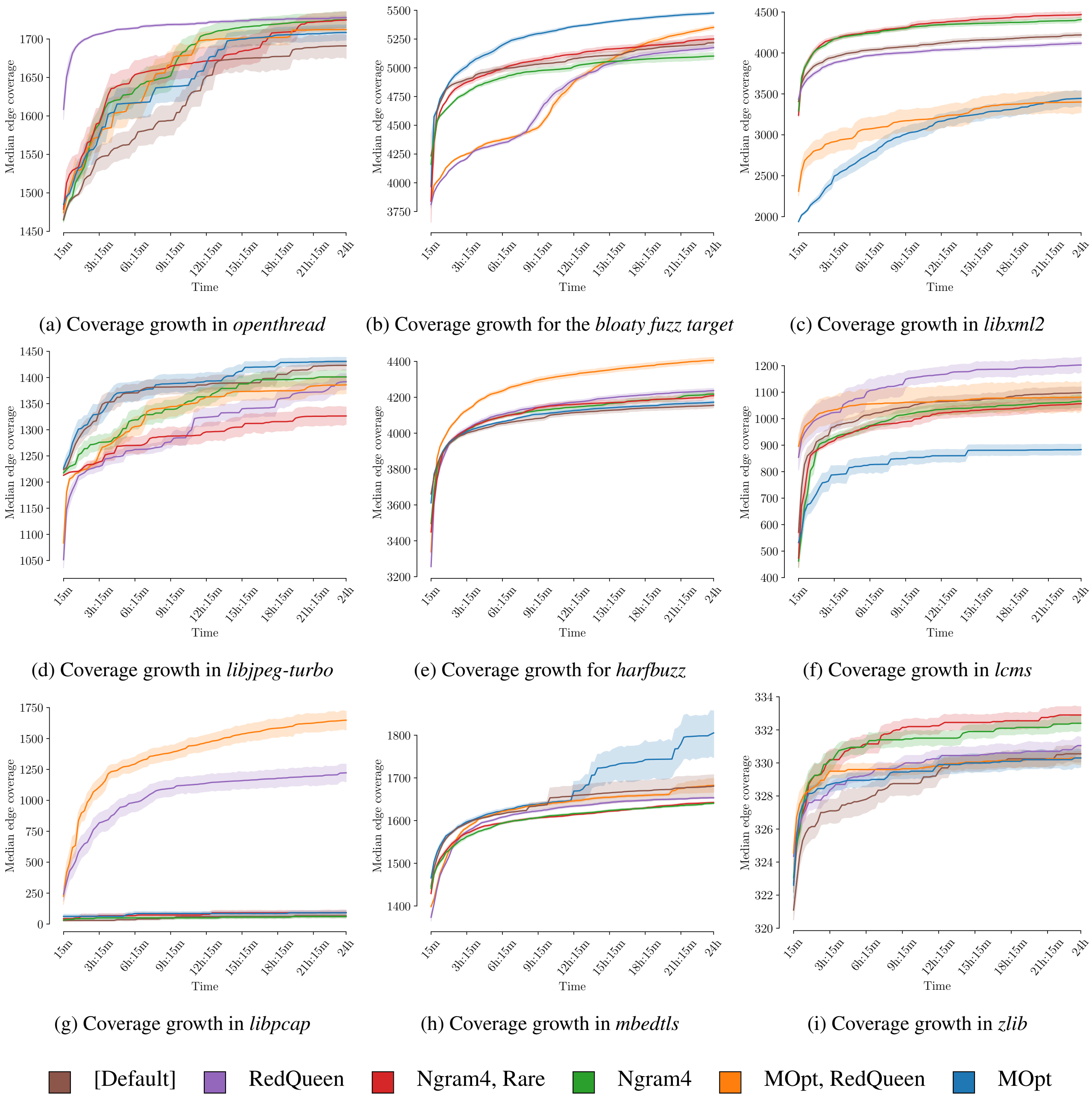
Based on our evaluations on FuzzBench, we performed hand-picked setups of AFL++ for a given target, in runs we call AFL++ *Optimal*.

This is an ongoing process, and will improve even further over time. The learning will, in turn, lead to better defaults in the fuzzer. For instance, SanitizerCoverage was found to be the best instrumentation option for most cases and therefore made the default.

We hand-crafted and enabled AFL++ Optimal for 13 of the 21 available targets. With this, the average normalized score of the percentage of the highest reached median coverage for each target increased by 7% and AFL++ Optimal generally outperforms all other fuzzers on FuzzBench[3].

[2] `https://www.fuzzbench.com/reports/2020-04-21-and-20-aflplusplus/index.html`

[3] `https://www.fuzzbench.com/reports/2020-06-30/index.html`



(a) Coverage growth in *openthread*

(b) Coverage growth for the *bloaty fuzz target*

(c) Coverage growth in *libxml2*

(d) Coverage growth in *libjpeg-turbo*

(e) Coverage growth for *harfbuzz*

(f) Coverage growth in *lcms*

(g) Coverage growth in *libpcap*

(h) Coverage growth in *mbedtls*

(i) Coverage growth in *zlib*

Median edge coverage growth with 66% confidence interval produced by each fuzzer over 20 trials at 24 hours.

5 Future Work

While the AFL++ project has reached good progress over the past year, there are bigger research and engineering problems yet to be addressed.

5.1 Scaling

Currently, AFL++'s scaling to multiple threads is less than ideal. Due to its decision to use the file system for test case delivery, at least for the backends that are not LLVM, and due to the reliance on the `fork()` syscall for certain targets, a larger part of the time is spent in the kernel. The development of the Linux Kernel Mode for snapshots is the first step in this direction. We made the AFL++ code fully thread-safe. The logical next step will multi-threading support, minimizing the overhead for synchronization between parallel fuzzers.

5.2 Collision-Free instrumentation

The original instrumentation offered by AFL hashes the current jump from and to addresses, in a way that potentially collides. This is seen as a trade-off between speed and accuracy, a problem that we strife to solve in the near future for both instrumentations based on source code and on emulation.

5.3 Static Analysis for Optimal Fuzz Settings

We do the research presented in this paper to optimize the presets of AFL++. The current goal is to use the most commonly best instrumentation, mutation and scheduling as the default configuration. However, as we showed in Sect. 4, the optimum depends *heavily* on the target.

For future work, finding indications for these through prior static analysis of the target could suggest a best-effort optimum solution — for example a lot of `strcmp` could be an indication to use RedQueen and others. This work will be based on the results of AFL++ *Optimal*, discussed in Sect 4.1.

5.4 Plug-in System

While Custom Mutator already grants large flexibility to researchers, the goal is to add additional plug-in functionality to replace or add functionality to building blocks such as schedulers, executors and queues.

Additional feedbacks in addition to hitcounts coverage will be supported, reimplementing from scratch the ideas shown in [42] [32] [4].

6 Conclusion

The tool discussed in this paper, AFL++, tries to integrate many of the major fuzzing research of late, where this is feasible to integrate in the current AFL++ architecture and our own benchmark show that there is a real-world improvement of the technique - at the least for corner cases.

After benchmarking all implemented functionality on a clear playing field, as laid out in Sect. 4, it becomes apparent that this is an important step for fuzzing research and development. Each proposed optimization shines for specific targets while it may perform less than ideal for others. the evaluation shows this is a clear benefit, for performance, as well as for comparability of research.

On top, AFL++ bridges the gap between academia and industry, making academic advancements available to everybody in an easy-to-use fashion. With the help of AFL++, a wide variety of real-world bugs could already be uncovered and patched, such as the bugs found by the community and listed in Table 2.

An excerpt of public bugs found by the community using AFL++

Program	Bugs	Discovered by
VLC	CVE-2019-14437 CVE-2019-14438 CVE-2019-14498 CVE-2019-14533 CVE-2019-14534 CVE-2019-14535 CVE-2019-14776 CVE-2019-14777 CVE-2019-14778 CVE-2019-14779 CVE-2019-14970	Antonio Morales
Sqlite	CVE-2019-16168	Xingwei Lin
Vim	CVE-2019-20079	Dhiraj
Pure-FTPd	CVE-2019-20176 CVE-2020-9274 CVE-2020-9365	Antonio Morales
Bftpd	CVE-2020-6162 CVE-2020-6835	Antonio Morales
Tcpdump	CVE-2020-8036	Reza Mirzazade
ProFTPd	CVE-2020-9272 CVE-2020-9273	Antonio Morales
Gifsicle	Issue 130	Ashish Kunwar
FFmpeg	Tickets 8592, 8593, 8594, 8596	Andrea Fioraldi
Glibc	Bug 25933	David Mendenhall

Over the course of more than a year, we were able to gain further insights into fuzzing thanks to the work put into AFL++. This provided us with the opportunity to fine-tune the AFL++ parameters in AFL++ *Optimal*, showing the power of the toolbelt in AFL++, available to everyone. With AFL++ as a platform, we hope to give researchers a quick and easy start to prototype and implement new ideas and strategies. The *Custom Mutator API* plug-in system makes it easy to prototype new research ideas and offers industry professionals an easy way to tailor test cases to their target, while still profiting from academic research and features. Using its proxy, it can even be adapted to completely new targets, servers, embedded targets, and more. Hopefully, future research can directly be based on AFL++'s APIs, further improving the state-of-the-art.

In conclusion, we invite researchers to contribute to the growth of AFL++ itself as a tool of interest for the community. AFL++ is — and always will be — Free and Open Source Software.

Acknowledgements We want to firstly thank the entire community around AFL++ that during this year actively contributed with patches, bugfixes and new features. Thanks also to Jonathan Metzman, Abhishek Arya, and László Szekeres for FuzzBench and the support provided to run AFL++ evaluations on it.

References

[1] Wine website. https://www.winehq.org/.

[2] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016.

[3] Tiny Code Generator (TCG). https://wiki.qemu.org/Documentation/TCG, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, 2018.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] Andrea Biondo. Improving AFL's QEMU mode performance. https://abiondo.me/2018/09/21/improving-afl-qemu-mode, 2019.

[9] Marcel Böhme. Entropic: Boosting LibFuzzer Performance. https://reviews.llvm.org/D73776, 2020.

[10] Marcel Böhme, Valentin Manès, and Sang Kil 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*, ESEC/FSE, pages 1–11, 2020.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *CoRR*, abs/1803.01307, 2018.

[13] S. Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE S&P 2020*, 2020.

[14] M. Eddington. Peach fuzzing platform. http://community.peachfuzzer.com/WhatIsPeach.html.

[15] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 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, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, 2020.

[17] Ivan Fratric. WinAFL. https://github.com/googleprojectzero/winafl, 2016.

[18] Marc Heuse, Heiko Eißfeldt, Andrea Fioraldi, and Dominik Maier. AFL++ Documentation. https://aflplus.plus/docs/, 2020.

[19] Christian Holler. Holler's AFL fork. https://github.com/choller/afl.

[20] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. 2018.

[21] Tessier C. Hubain C. Implementing an LLVM based dynamic binary instrumentation framework. https://qbdi.quarkslab.com/QBDI_34c3.pdf, 2017.

[22] László Szekeres Jonathan Metzman, Abhishek Arya. FuzzBench: Fuzzer benchmarking as a service. Google Security Blog, March 2020.

[23] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[24] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[26] Dominik Maier, Benedikt Radtke, and Bastian Har-

ren. Unicorefuzz: on the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[27] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: BasebandSAnitized Fuzzing through Emulation. In *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 20)*, Linz (Virtual Event), Austria, July 2020.

[28] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *arXiv: Cryptography and Security*, 2018.

[29] Antonio Morales. Fuzzing software: common challenges and potential solutions (Part 1) - GitHub Security Lab. https://securitylab.github.com/research/vlc-vulnerability-heap-overflow, 2020.

[30] Anh Quynh Ngyuen and Hoang Vu Dang. Unicorn: Next generation cpu emulator framework, 2020.

[31] Tavis Ormandy. Making Software Dumberer. http://taviso.decsystem.org/making_software_dumber.pdf, 2009.

[32] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[33] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, May 2018.

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

[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[36] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. Kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 167–182, USA, 2017. USENIX Association.

[37] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP 2010, pages 317–331, 2010.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28. USENIX Association, 2012.

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

[40] Nathan Voss. afl-unicorn: Fuzzing arbitrary binary code, October 2017.

[41] Guido Vranken. libfuzzer-ng: enhanced fork of libFuzzer. https://github.com/guidovranken/libfuzzer-gv, 2017.

[42] Guido Vranken. VrankenFuzz a multi-sensor, multi-generator mutational fuzz testing engine. https://guidovranken.files.wordpress.com/2018/07/vrankenfuzz.pdf, 2018.

[43] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu 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)*, pages 1–15, Chaoyang District, Beijing, September 2019. USENIX Association.

[44] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512, May 2010.

[45] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2313–2328, New York, NY, USA, 2017. Association for Computing Machinery.

[46] B. Yadegari and S. Debray. Bit-level taint analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 255–264, 2014.

[47] Michał Zalewski. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016.