Fuzzing: Ideas, Roadblocks, and Strategies

Matthis Gördel
TU Berlin
matthis.goerdel@tu-berlin.de

ABSTRACT

Fuzzing – automated testing with randomly generated inputs – is an essential tool to test and secure important low-level components of our modern digital infrastructure. The term was coined in the early nineties and the method proved effective in crashing UNIX utilities. A wave of innovation in the field was sparked in the mid-2010's by the seminal *American Fuzzy Lop* (AFL), a highly pragmatic fuzzer. Its main contribution was to make coverage-guided greybox fuzzing – i.e. fuzzing that focuses on mutating inputs that previously discovered new execution traces – practical by making the tool fast and usable. In the following years, many researchers extended AFL or its approaches by improving key algorithms, adding heuristics to surpass roadblocks like *magic values*, or combining greybox-fuzzing with more sophisticated approaches like symbolic execution. But with this research output comes the question of how to evaluate the different approaches in a fair and meaningful way.

KEYWORDS

ACM proceedings, Fuzzing, Testing, Reliability

1 INTRODUCTION

Heartbleed [2] is one of most consequential security vulnerabilities to date [11]. The vulnerability occurred in OpenSSL, an important library that is used in many internet-facing services. Interestingly, Böck argues that Heartbleed could have been avoided (i.e. the bug being found before hitting production) by *fuzzing*, i.e. testing OpenSSL with automatically generated inputs: In 2015, Böck demonstrated that it would take the AFL fuzzing engine roughly six hours to find the vulnerability. His main takeaway was that fuzzing should be used to detect similar lingering vulnerabilities [9].

The term *fuzzing* originates from a dark and stormy night in 1988, where interference of a dial-up connection caused scrambled inputs which in turn crashed core system utilities. Motivated by the occurrence, a systematic testing of UNIX utilities with random inputs was facilitated, finding bugs in programs like `uniq`, `make` and `csh` [21].

Today, more than thirty years later, fuzzing is a vital part of testing and securing critical software with a vibrant research community, even though random inputs still crash UNIX utilities [22].

In this paper, we want to give a general overview over the past and present of the field. In Section 2, we introduce general concepts of fuzzing, explain terminology, and present a model for the fuzzing process. Then, in Section 3, we present AFL, a popular and influential fuzzer that sparked a lot of innovation in the field. In Section 4, we focus on said innovation, highlighting approaches that were either implemented on top of AFL or tackles problems that are also relevant for AFL. Finally, we cover challenges of fuzzing research in Section 5, highlighting the complexities of evaluating fuzzers and obtaining definitive results.

CARE'23, Summer Term 2023, Berlin, Germany
.

2 A GENERAL FUZZER

In this section, we propose a simple fuzzer and extend it piece by piece to introduce concepts and terminology.

First, we need a program that we want to test. For the beginning, we assume a program that consumes its input via standard in. We call this program the *program under test*. To find bugs in the program under test, our fuzzer needs to generate inputs, to run the PUT with the inputs and to detect crashes or hangs.

2.1 Generating Inputs

In [21], Miller et al. used a small program called `fuzz` to generate random input with optional restrictions like *only printable characters* and a shell script to execute the program under test. Crashes are detected by checking for the existence of *core files* that the operating system creates when a program crashes. As demonstrated by Miller et al., it is possible to find actual bugs with this approach, but there are more efficient and effective approaches.

For example, instead of *generating* input completely at random, a well-structured input can be used as a starting point and then be *mutated* at random. We call such a file a *seed* and the collection of seeds the *corpus*. We call fuzzers that work by mutating seeds *mutational*, as opposed to *generational* fuzzers that generate inputs without using seeds. This helps with fuzzing programs that expect a certain structure in the input, like e.g. a media player that expects the input file to have some specific header, since the mutated seed is mostly correct as opposed to completely random input, which is mostly incorrect and might be rejected early by the PUT. Zzuf [14], initially released in 2006, is a fuzzer that leverages this technique. It executes a program like the VLC media player with its input, e.g. a mp4-file, randomly mutates that file and detects crashes of the program under test.

A third option to generate inputs is to define a grammar that states how the input shall be structured. The fuzzer then generates random inputs that adhere to the grammar. We sometimes call these structure-aware fuzzers *smart*. Listing 1 shows a grammar used by the PEACH fuzzer [3] that describes a part of the WAV audio format.

2.2 Detecting Errors: Sanitizers

With the ability to generate/mutate inputs, we now focus on detecting errors. The fuzzer of Miller et al. simply looked for *core dumps*[1] to detect erroneous executions. While this method has no false positives, it might miss some program failures that are less violent. For example, a program could encounter undefined behavior, an out-of-bounds memory access or a data race, maybe even

[1] Files created by the operating system after handling a crash of the program.

Listing 1: PEACH grammar describing a part of the WAV format (Copied from [24])

```
<DataModel name="Wav" ref="Chunk">
  <String name="ckID" value="RIFF"/>
  <String name="WAVE" value="WAVE"/>
  <Choice name="Chunks" maxOccurs="30000">
    <Block name="FmtChunk" ref="ChunkFmt"/>
    ...
    <Block name="DataChunk" ref="ChunkData"/>
  </Choice>
</DataModel>
```

corrupting other files on the system, all without crashing. Since for a crash to occur, the program would have to misbehave in such a way that the operating system recognizes, e.g. dividing by zero or accessing a memory region that belongs to another process.

In this situation, sanitizers can be helpful. Sanitizers are dynamic bug finding tools, widely used in testing C/C++ code and a research field on their own. They work by adding extra checks and assertions during compilation. During program execution, these checks search for incorrect program behavior as it happens [32].

For example, AddressSanitizer (ASan) [29] works by keeping metadata about which memory regions are accessible and replaces the `malloc` and `free` implementations so that the metadata is kept up to date. At each memory dereference, the metadata is checked to confirm that the program actually allocated the memory. While this causes a slowdown that roughly doubles the execution time of the program [28], it can help catch crucial bugs. Other sanitizers are e.g. *ThreadSanitizer* (TSan) [30], *Undefined Behavior Sanitizer* (UBSan) and *Effective Type Sanitizer* (EffectiveSan) [10].

To leverage a sanitizer for fuzzing, we compile our program under test with the sanitizer enabled, and then fuzz it. The additional checks introduced by the sanitizer turn previously silent errors like undefined behavior into crashes, which then are detected by the fuzzer.

2.3 Greybox Fuzzing

With our different input-generating methods in place and also being able to detect crucial types of errors, our fuzzer can already find bugs, but it is still working *randomly*, without any plan or understanding of the program under test besides rough knowledge of the shape of the input. This is where *greybox fuzzing* steps in and revolutionizes fuzzing: We add some more instrumentation to our program under test, similar to how we added the fuzzers, but this time the instrumentation is not for detecting errors but rather to track the code coverage, i.e. which lines of code are executed when running the PUT with the input. This information can then be leveraged to find interesting directions in which to mutate the input.

Assuming we fuzz a media player like VLC by mutating some mp4 file and also track code coverage: First our fuzzer mutates e.g. some bits at the end of the file and runs the program with the inputs, but no new lines covered compared to the execution with the initial seed. Thus, the fuzzer decides to mutate another region of the seed file. Now new lines are covered, thus the fuzzer maybe decides to mutate that region a bit more, in the hope of generating more input that covers new lines. This approach is called *greybox fuzzing* since the fuzzer peeks a little into the program under test by instrumenting for coverage. The earlier approach without this coverage feedback is correspondingly called *blackbox fuzzing*.

Listing 2: Highly specific branch conditions can be handled by whitebox fuzzers that track and solve the condition while non-symbolic approaches generally struggle here.

```
def foo(x: int64):
    if x == 0xd9ce994a2f133f02:
        return 1 / 0
    return 1
```

2.4 Whitebox Fuzzing

When there's grey and black, there also has to be white: *Whitebox fuzzing*, a term for the related approaches of *symbolic execution* and *concolic execution* involves not only tracking code coverage but tracks the logical paths through the program. Consider the function shown in Listing 2.

Except for a case of exceptional luck, a black- or greybox fuzzer would have to fuzz the function many times to finally guess the magic value to hit the path that leads to the crash. But a whitebox fuzzer would execute the function at most twice.

During symbolic execution, the program is not run with concrete values (e.g. 42) but with *symbolic* values that capture path constraints (e.g. $x > 42$). So when encountering the branch statement in Listing 2, the symbolic execution engine forks the execution, once entering the `then` case and adding the constraint $x =$ `0xd9ce994a2f133f02`, once entering the `else` case and adding the constraint $x \neq$ `0xd9ce994a2f133f02`. While this prevents the necessity to re-execute paths of the program, the high overhead of symbolic interpretation and the constraint solving required to e.g. check which memory objects a symbolic pointer could point to, currently make this approach less practical than greybox fuzzing.

Besides symbolic execution, there is *concolic execution*. Concolic is a portmaneau of *concrete* and *symbolic*. During concolic execution, the program is run with a concrete input like 42, but additionally, the path constraints are tracked. After finishing the execution, one path constraint is picked and negated. Then, a constraint solver is used to generate a new concrete input that satisfies the new constraint set, thus reaching the branch not yet taken that corresponds to the constraint that was negated. So when a concolic execution engine runs the code depicted in Listing 2, for the first time, x might be 42, thus the `else` branch is taken. But in addition, the path constraint $x \neq$ `0xd9ce994a2f133f02` is collected. After the run, the engine might pick the just collected constraint to be negated by the solver, thus generating `0xd9ce994a2f133f02` as the next input. While concolic execution has the advantage of a much lower execution overhead compared to symbolic execution, common paths at the beginning have to be re-executed, just like in greybox fuzzing.

While there is a lot of research involving whitebox fuzzing, this paper will mostly focus on coverage-based greybox fuzzing, which is currently – in the sense of industry adoption – the more popular of the approaches, which might indicate a higher effectiveness. Note that there is another paper in the CARE seminar focusing solely on whitebox fuzzing.

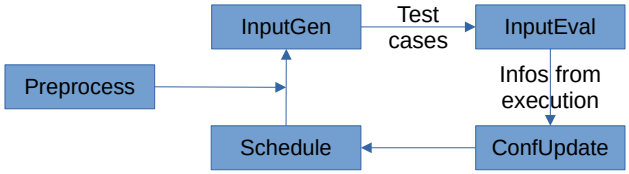


Figure 1: A general model for a fuzzer (Adapted from [19]). First, the program under test is preprocessed (e.g. instrumented to track coverage). Then input is generated, e.g. by choosing a seed from the corpus and mutating it. Next, the PUT is run with the input. Then, information about the run (e.g. newly covered lines) are used to update the state of the fuzzer. This then influences the scheduling decision, i.e. which seed to choose next.

Table 1: Different fuzzing approaches (Copied from [7]). The approaches are distinguished by whether program analysis is used and whether additional feedback from the execution besides if it crashed or not is obtained.

Blackbox fuzzing	no analysis, no feedback
Greybox fuzzing	no analysis, but coverage feedback
Whitebox fuzzing	mostly program analysis

2.5 Summary: A Fuzzing Model

After introducing different fuzzing concepts and terms, we summarize and formalize our understanding. First, Table 1 recaps the distinction between black-, grey-, and whitebox fuzzing.

Next, we want to formalize our understanding of the fuzzing process. Manès et al. [19] suggest an abstract model for fuzzing, depicted in Figure 1 with the following steps, in our case simplified for greybox fuzzing:

- **Preprocess**: In which the program under test is instrumented to e.g. track coverage, add sanity checks and/or analyzed e.g. to gather extra information for seed scheduling. as done in [7] or to e.g. extract magic values as done in [25].
- **Schedule**: In which the next seed is selected.
- **InputGen**: In which the seed is mutated e.g. by flipping bits.
- **InputEval**: In which the program under test is run with the mutated input.
- **ConfUpdate**: In which a new seed might be added if the previous mutated seed had some interesting property like covering new parts of the program.

3 DESIGN AND ARCHITECTURE OF AFL

Fuzzing not only a vibrant academic field but also leveraged by industry. While the term was coined in academia, many important fuzzers were created outside of academia, simply to find bugs. The seminal *American Fuzzy Lop* (AFL) was created by Zalewski at Google around 2013, with the explicit goals of speed, reliability and simplicity, focusing on usefulness rather than novelty [34]. In this section, we give an overview over its design and architecture.

Figure 2: AFL's core algorithm. (Adapted from [35]).

(1) Load user-supplied initial test cases (seeds) into a queue.
(2) Take next seed from the queue.
(3) Attempt to trim the seed to the smallest size that doesn't alter the measured behavior of the program,
(4) Repeatedly mutate the seed by flipping bits, setting bytes to special values like 0, `INT_MAX`, adding and removing bytes, etc.
(5) If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new seed in the queue.
(6) Go to 2.

AFL is a mutational greybox-fuzzers, i.e. it generates new inputs by mutating seeds and tracks coverage information of each execution. Its basic algorithm is shown in Figure 2. To gather coverage information, AFL offers a compiler wrapper that adds coverage instrumentation by injecting roughly the following code at the beginning of each basic block[2]:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

This allows to trace the execution, i.e. it tracks not only which lines of code are covered but also in which order. Note that collisions or overflows can occur, which makes this tracing method imperfect. Still, the instrumentation requires very little overhead, since there is e.g. no dynamic allocation and the coverage map is sized compact enough to fit into L2 cache. The potential imprecision of the tracing is traded off with the speed gained by the approach.

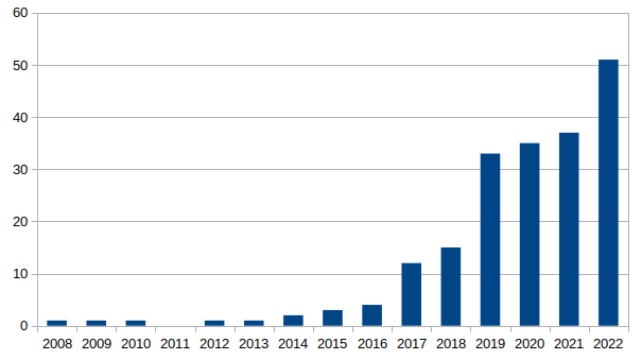
Using the coverage information, AFL searches for inputs that trigger previously unseen basic block transitions. Those inputs are then added to the input queue so that they later become starting points for further fuzzing, i.e. that they become new seeds. This way, AFL creates a corpus of inputs that trigger new behavior. Periodically, AFL prunes this corpus, distilling it to a subset that still covers the same behavior. The pruning also uses a simple algorithm: AFL rates each input according to execution latency and file size and then picks for each basic block transition the best-rated input. For file mutation strategies, AFL flips bits and bytes, increments and decrements integers or sets integers to interesting values like 0, 1, 256, `MAX_INT` etc. While AFL has more features, this brief overview should give the impression that AFL is very pragmatic, well-engineered software that focuses on getting the job done, disregarding novelty for usability. Nevertheless, or maybe because of this, many research projects based their implementations on AFL.

4 INNOVATION SINCE AFL

AFL was released in November 2013 and developed continuously until 2017. While initially not widely known, it received major attention after finding to bugs that circumvent the initial patch for

[2]A basic block is a code sequence without any branches or other control flow statements, it has only one entry and one exit point. The programs control flow can be modeled as a graph with basic blocks as nodes and the jumps etc. as edges.

Figure 3: Estimation[4] of fuzzing papers published in major conferences per year.



the shellshock vulnerability [9]. With time, the tool found many[3] critical bugs in software like GnuPG, OpenSSH, etc., further proving its effectiveness. But the tool was not only receiving attention from industry and practitioners aiming to find bugs in their products but also from researchers that aim to push the capabilities of fuzzing.

While the term fuzzing was coined in the early nineties [21], research interest increased around the 2015 mark, as shown by the number of publications in Figure 3. Thus, a few years after AFLs release, and often explicitly influenced by it, a wave of innovation in mutational greybox fuzzing started. In this section, we want to highlight some of these innovations. We focus on influential approaches that develop or extend a fuzzer, highlighting the key ideas. Table 2 gives an overview of the papers/work that we present.

Our selection of papers is definitely not comprehensive. Due to the scope of this paper we focus on approaches that fall fit a few themes, namely *using theory for efficiency* and *tackling magic numbers*. This is the reason why we for example ignore interesting approaches like kAFL [27], fuzzing counter measures [13][16], and much more. We recommend Manès et al. [19] for a much broader overview.

4.1 Using Theory for Efficiency

In this section, we focus on papers that analyse a certain aspect (namely: seed scheduling, mutation, mutation scheduling) of AFL's algorithm (c.f. Figure 2), then discover some theoretical properties which then are leveraged to increase the efficiency/capability of AFL.

4.1.1 AFLFast. Böhme et al. [8] improved AFL's seed scheduling algorithm by leveraging insight from probability theory. AFL maintains a queue of seeds. It picks a seed from the queue, mutates and executes the PUT with it a fixed number of times, and then appends that seed to the end of the queue. If new paths are discovered during execution, new seeds are also added to the queue. We call the number of times the seed is mutated when picked from the queue the *energy* assigned to the seed. The fact that AFL executes each seed the same amount of times can be expressed as AFL assigning each seed the same energy. The alternative would be to assign higher energy to certain seeds, i.e. mutate and execute these seeds more than other seeds when picked from the queue.

The key observation of Böhme et al. was that AFL's approach of *same energy for each seed* has two major drawbacks: It slows down the discovery of new seeds, and it focuses the execution on *hot* paths, i.e. paths that are already well-explored are more likely to be explored further.

The intuition for the first problem is the following: The probability to detect a new path and thus generating a new seed is not equal for all seeds. Thus, we first execute a seed only a few times and then gradually increase how often we execute the seed. Or, rephrased in with energy-terminology: We initially assign low energy to each new seed and gradually increase it over time. A metaphor for the approach could be a strategy for finding Easter eggs: First we search every room/spot only briefly, looking for simple caches, but with time we intensify our inspection of every room. But we do not look into every cupboard in the kitchen before at least glancing into the living room.

Böhme et al. implemented their improvements to AFL's seed scheduling in a fork called AFLFast, which was able to find same bugs multiple times faster.

AFLFast highlights how a thorough understanding of the fundamental theory can greatly improve a practical approach. The research also re-influenced AFL as Zalewski implemented improvements based on it [37].

4.1.2 AFLGo. After AFLFast, Böhme et al. [7] added the capability to *direct* the execution of the fuzzer, i.e. that the fuzzer focuses on covering a certain location in the program. This can be useful for patch testing or crash reproduction when given a stack trace. Directed fuzzing was at the time already possible with whitebox fuzzing. For this, constraint solving was used to determine which inputs are required to reach the target location [20][15]. The drawback of this approach is its high overhead required to track and solve constraints. Since greybox fuzzers are currently much more useful in practice due to their efficiency, it was desirable to implement directed fuzzing in greybox fuzzing. For this, Böhme et al. calculate the approximate distance of each basic block to the target location when compiling and instrumenting the program under test. To steer the execution, the energy-based approach from AFLFast is reused, but instead of giving higher energy to seeds that execute rather unexplored paths, higher energy is assigned to seeds that reach closer to the target. The approach outperforms the previous symbolic execution-based approaches.

4.1.3 AFLSmart. While AFL and comparable greybox fuzzers generally outperform *smart* blackbox fuzzers like peach, that use a grammar to generate inputs that comply with the high-level structure of the format [24], there is still benefit in giving AFL such additional information when fuzzing applications that handle highly structured inputs like PDFs, PNGs, etc. Pham et al., who also developed AFLFast and AFLGo, extended AFL so that it can leverage the input structure specifying grammars of the peach fuzzer mentioned in the introduction. They add new mutation operators, so that the fuzzer mutates the input on the chunk level, i.e. mutating, duplicating, and deleting chunks instead of bytes.

[3] The AFL website [36] lists 370 "notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL".
[4] Data gathered from dblp [1] of papers from the conferences NDSS, S&P, ICSE, USENIX Security Symposium, ASE, CCS, and ESEC that have "fuzzing" in their title.

Table 2: Notable Developments in Fuzzing since the release of AFL.

Name	Conference	No. Citations*	Main contribution
Driller	NDSS 2016	986	tackle magic numbers by concolic execution
AFLFast	CCS 2016	786	more efficient exploration via seed scheduling
laf-intel	Blogpost 2016	7	tackle magic numbers by splitting comparisons via compiler pass
Vuzzer	NDSS 2017	662	tackle magic numbers by extracting `cmp` immediates, taint analysis
AFLGo	CCS 2017	587	directed fuzzing via seed scheduling
T-Fuzz	SP 2018	309	tackle magic numbers by patching out roadblocks
REDQUEEN	NDSS 2019	244	tackle magic numbers by input-state correspondence
AFLSmart	ICSE 2019	160	tackle structured inputs via PEACH grammars
MOPT	USENIX 2019	189	more efficient exploration via mutation operator scheduling
AFL++	USENIX 2020	261	AFL-fork that merges AFLFast, MOPT, REDQUEEN, etc.

*As reported by Google Scholar.

4.1.4 MOPT. Scheduling is an important part of the execution of AFL. AFLFast discovered that it is beneficial to focus on exploring seeds that discover more new paths. But not only seeds are scheduled during the execution, but also the mutation operators, i.e. in which way the seed is mutated (e.g. bit-flip, setting a byte to 0, etc.). Lyu et al. discovered that most new paths are discovered by certain mutation operators like the single bit-flip. To leverage this fact, they implement MOPT, which optimizes the scheduling of mutation operators.

4.2 Magic Numbers and Other Roadblocks

A roadblock hinders a fuzzer from executing certain parts of the PUT. Roadblocks arise due to the concrete approach of the fuzzer, thus different fuzzers have different roadblocks. The exploration of a PUT that expects highly structured inputs by a mutational fuzzer might be blocked input file validations, while a smart fuzzer with e.g. a grammar describing the input format can overcome these hurdles.

Since AFL mutates inputs randomly, it struggles with passing checks that require very specific inputs. The simplest case is handling *magic bytes* (in this case: "fixed sequence of bytes at a fixed offset in the input") as e.g. shown in Listing 3. If AFL does not have a seed that reaches the `then` block, it has to guess the correct values out of the roughly four billion possible 32-bit integers.

Checksums are even more problematic: here, there is no fixed value that has to be guessed and might be inferred with some clever trick, but the specific value depends on the supplied input.

There are different approaches on how to handle these hurdles, which we will now present.

4.2.1 Driller. Stephens et al. [33] extends AFL by periodically leveraging *concolic execution* to generate new inputs that reach new code sections. Thus, it blends grey- and whitebox fuzzing. The key intuition of the approach is that the program under test consist of *compartments* separated by difficult to pass, highly-specific checks like checks for magic numbers. To pass these checks, Driller suspends AFL-fuzzing from time to time and instead hands the current seeds to *angr* (c.f. [31]), the concolic execution engine. Angr executes the program under test with these seeds and tracks the path constraints at the encountered branches. It then leverages a constraint solver to generate new seeds that reach the previously unexplored path of the branch statement.

Listing 3: Magic number hindering program exploration (Adapted from [5])

```
if (input[42] == 0xabad1dea) {
  /* difficult to reach by fuzzer */
} else {
  /* other code */
}
```

Listing 4: Code from Listing 3 transformed by laf-intel (Adapted from [5]).

```
if (input[42] >> 24 == 0xab){
  if ((input[42] & 0xff0000) >> 16 == 0xad) {
    if ((input[42] & 0xff00) >> 8 == 0x1d) {
      if ((input[42] & 0xff) == 0xea) {
        /* much less difficult to reach by fuzzer */
}}}} else {
  /* secure code */
}
```

4.2.2 laf-intel. Laf-intel (c.f. [5]) is, just like AFL, an example of non-academia innovation. It is not a fuzzer or fuzzer-extension but a collection of LLVM passes. An LLVM pass describes how to transform parts of a program, e.g. how to simplify the statement x+1+1 to x+2. LLVM passes are leveraged by LLVM-based compilers like clang. The laf-intel pass splits up magic value-comparisons in such a way that the values are easier to guess for AFL. So the four-byte comparison shown in Listing 3 is transformed into four one-byte comparisons as shows in Listing 4. This way, AFL only has to guess a single byte each time, which increases the probability to reach the desired location from 256^{-4} to $4 \cdot 256^{-1}$.

4.2.3 Vuzzer. Rawat et al. [25] implemented Vuzzer which tackles the magic-value problem by analyzing the program under test before execution, during which hard-coded values (i.e. *immediate* parameters of `cmp` instructions) are extracted, like `0xabad1dea` in the example shown in Listing 3. During execution, the data flow of the program is analyzed, to determine which input bytes are compared against the immediate values are compared against. In

this example, the analysis is trivial assuming that `input` is the raw input. With this analysis, Vuzzer would then generate a new input that sets four bytes at the corresponding offset to the immediate value that was extracted earlier.

Compared to Driller, Vuzzer chooses a less general, more heuristic-focused approach, since only comparisons against immediates are handled.

4.2.4 T-Fuzz. Peng et al. [23] implemented T-Fuzz which tackles the magic-value problem in a quite radical way: When it determines that a specific check is blocking fuzzer progress, it simply removes the check by patching it out. When T-Fuzz finds a bug in the patched program, it determines the additional input constraints from the patched-out checks. If the constraint set is satisfiable, an input that reaches the bug is generated, if it is not satisfiable, the bug in the patched program was impossible, and thus a false positive. Lastly, the unpatched program is run with the generated input to fully verify that the bug was no false positive.

4.2.5 REDQUEEN. Aschermann et al. [4] implemented REDQUEEN which tackles the magic-value problem by generalizing the approach of Vuzzer. It leverages the key observation that often parts of the input correspond (quite) directly to the state of memory at runtime, i.e. that e.g. the bytes `0xdeadbeef` encountered in memory at runtime were supplied as input (as opposed to calculated at runtime).

For this, the inputs of `cmp` instructions are instrumented to record actual and expected values, e.g. `0x12345678` and `0xcafebabe`. During the next fuzzing iteration, the previously recorded actual values (e.g. `0x12345678`) are replaced with the corresponding expected values (e.g. `0xcafebabe`).

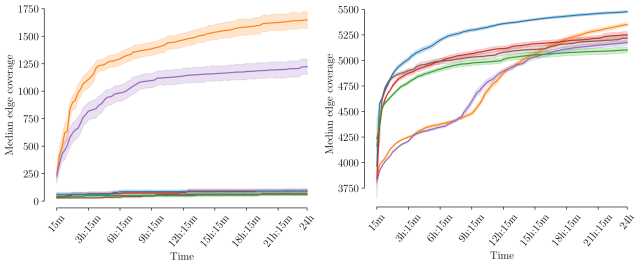
This approach has two advantages over Vuzzer: First, Vuzzer can only handle branches with hard-coded values, REDQUEEN can handle all branches where input and state correspond. Second, Vuzzer's data flow analysis has a higher overhead than REDQUEENs *instrument* `comp` *and search+replace in input* approach, making REDQUEEN more efficient.

4.3 AFL++: Shared Infrastructure for Further Research

Since continued development of AFL has come to a halt in 2017 even though there was still interest from community and academia in the project, Fioraldi et al. [12] decided to fork it as AFL++ to incorporate patches and improvements. Most notably, they incorporated the approaches of AFLFast, MOPT, laf-intel and REDQUEEN into AFL++. They also improved some engineering decisions like the mechanism how the PUT is executed.

Besides building an improved fuzzer, their goal was also to create a technical foundation for further fuzzing research. For this they added APIs to simplify the implementation of new mutation operators or seed scheduling strategies. This has the additional benefit of making cross-evaluations like the one shown in Figure 4 easier: If a research group implements a novel approach with AFL++, they can easily benchmark the approach against the state-of-the-art approach implemented in AFL++.

Figure 4: Results from Fioraldi et al.'s cross-evaluation [12]. Each line represents the coverage of AFL++ achieved over time with certain techniques dis- and enabled. The purple line represents REDQUEEN, the blue line represents MOPT, and the orange line represents REDQUEEN+MOPT. The difference between the two plots is the program under test. Notice that when fuzzing libpcap, REDQUEEN+MOPT performs best, but when fuzzing bloaty it performs worst for a long time. This highlights the complexities of comparing different techniques.



(a) PUT: libpcap **(b) PUT: bloaty**

4.4 A Genealogy of Approaches

There are several interesting connections between the different fuzzers, which we tried to visualize in Figure 6.

AFLFast introduces the idea of energy-based scheduling, which was incorporated into e.g. Vuzzer. The energy-based scheduling approach not only improved scheduling, but it also laid the foundation for *directed fuzzing*, which was implemented by changing the energy-assignment from focusing on *cold* paths to focusing on reaching towards the target location.

The idea of laf-intel to mutate progress-hindering `cmp` instructions was extended in T-Fuzz that completely removes such instructions.

Vuzzer also focuses on `cmp` instructions but instead extracts the magic values and uses data flow analysis to determine the corresponding input bytes. REDQUEEN improves this approach by removing the necessity of data flow analysis via the input-state-correspondence heuristic, significantly reducing overhead.

Also, T-Fuzz and Driller (which is based upon the concolic execution engine angr) both incorporate symbolic/concolic execution (thus the grey and white fill color since grey- and whitebox fuzzing are combined).

While this genealogy is far from complete, it hopefully shows how implementations lay the foundations for further research and how ideas are reapplied and recontextualized in different approaches.

5 CURRENT CHALLENGES IN FUZZING

In the previous section, we presented different approaches, that all aim to improve fuzzing. But it turns out that it is often not so simple to show that one approach is superior to another. In this section we want to focus on two intertwined challenges of fuzzing research: First, we highlight the difficulties in comparing different fuzzers against each other. Then, we point to the fact that

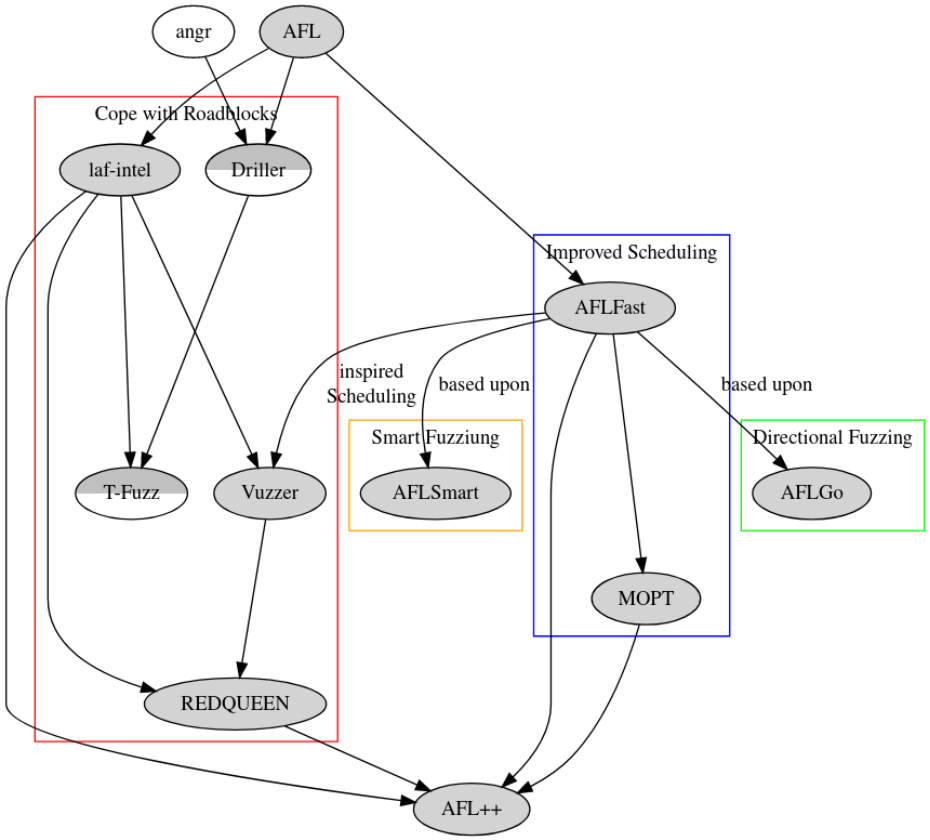


Figure 6: A genealogy of fuzzers building upon or influenced by AFL. A grey backgroud represents greybox fuzzing, a white background whitebox fuzzing. Two-colored nodes leverage both approaches. The rectangle subgroups aim to highlight the theme of contribution.

developing a fuzzer is a complex engineering endeavour, where (mis-)engineering can substantially influence – or, more pessimistically: skew – the outcome.

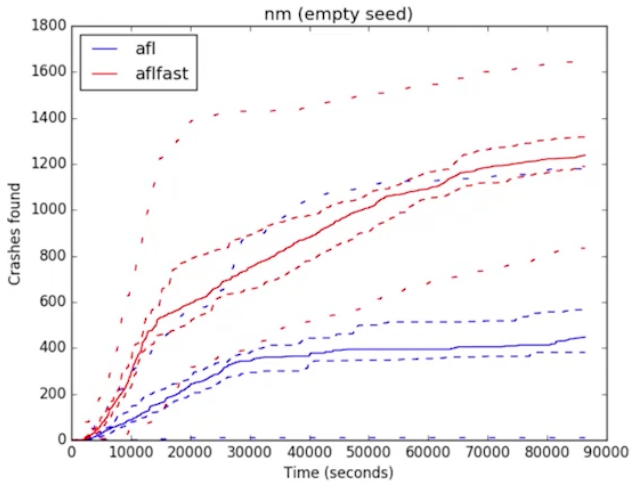
5.1 Comparing Fuzzers

Current fuzzing publications that propose new approaches generally evaluate their approach *empirically* by benchmarking their fuzzer. While empirical research is of course a good idea, evaluating complex systems like fuzzers has its inherent challenges. Klees et al. [17] and Li et al. [18] both highlight these challenges.

Klees et al. first propose the core parts of an evaluation: A baseline like plain AFL to compare against, different PUTs, one or more performance metrics like *no. of unique bugs found* of *line coverage*, and then conducting a sufficient number of trials. But with each of those parts come challenges: Differences between the baseline fuzzer and the to-be evaluated fuzzer can make for an uneven comparison, e.g. if the baseline fuzzer is implemented in an inherently slower programming language. If possible, it would be advisable to implement the new technique on top of a high-quality fuzzer like done by AFLFast, MOPT, etc. As already highlighted in Figure 4, the chosen PUT also strongly influences fuzzer effectiveness. Performance metrics can also be difficult: Simply counting the number of times the fuzzer was able to crash the PUT is obviously a bad idea, since this would reward a fuzzer that finds a single bug and then generates many inputs that trigger that bug. A much better metric would be the number of distinct bugs found by the fuzzer, but this of course requires the work of debugging and fixing the PUT until the fuzzer cannot find bugs anymore. Another good metric could be line coverage since a fuzzer that can surpass more hurdles/roadblocks than the baseline should be able to translate this advantage into more lines covered. Another big problem is the statistical significance of the results: Klees et al. highlighted that multiple runs of the same fuzzer with the same PUT can have a high variance, as shown in Figure 7. Thus, it is important to not only have one run or simple calculate the mean of all runs but to also mention confidence intervals.

To tackle these challenges, Klees et al. recommend to use multiple trials in combination with statistical tests to show significance, using a diverse set of PUTs and long timeouts. They identify the curation of a suite of PUTs for benchmarking as an important area of future work. Li et al tried to create such a benchmark suite with Unifuzz [18] but it seems that their proposed framework didn't have a lasting impact since the github repositories are stale since roughly three years.

Figure 7: Variance of different evaluation runs (Copied from [17]). The solid lines show the mean of all runs, the densely dashed lines show confidence intervals and the sparsely dashed lines show the minimum and maximum values recorded. We see that AFLFast might outperform AFL is some runs by a big margin, but in other runs AFL outperforms the mean of AFLFast's performance.



5.2 Idea vs. Implementation

Böhme et al. [6] summarized the discussions of researchers and practitioners regarding current challenges in fuzzing. One interesting topic, especially in the context of this paper, is the difficulty of evaluating and comparing different fuzzers. While it is even difficult to compare two fuzzers where one is a direct extension of the other (c.f. [17]), it is even more difficult when the fundamental implementations differ. As Rizzi et al. [26] demonstrated by reviewing extensions of the KLEE symbolic execution engine (a popular whitebox fuzzer), the improvement gained from a novel technique can be overshadowed by improvements gained simply from fixing mis-engineered parts of the base-tool like a broken cache.

While greybox fuzzers have a much higher practical adoption than KLEE and other symbolic execution tools, this still puts a burden on the implementor of novel techniques to ideally match the high quality, efficient code written by Zalewski and others to not skew research findings. This also makes comparing ideas really difficult: Is A a better technique than B since it outperforms it by a factor of x, or was A simply profiled more thoroughly?

The discussion around FidgetyAFL [37] might be a prototypical example of discussions about *implementation* vs. *idea* and the difficulty of proving that one approach is better than the other.

5.3 Other Challenges

Böhme et al.[6] give a great overview of other current challenges to fuzzing that do not fit the theme of this paper like the problems involving stateful systems of exotic targets like microcontrollers.

6 CONCLUSION

In this paper, we aimed for a broad overview of recent research in fuzzing, focusing especially on greybox fuzzing, AFL and the research inspired and built on top of it.

We highlighted how Zalewski created with AFL a solid foundation that turned out to be a great facilitator of research. We then highlighted how different research endeavors built on top of each other. Furthermore, we contributed a sketch of a genealogy of AFL in which we showed the connections between different research projects.

Lastly, we focused on the current challenges in the field, namely the intricacies of evaluating fuzzers and the dependence on systems engineering which requires extensive implementation effort to try out new ideas and makes comparisons of those challenging.

REFERENCES

[1] [n. d.]. DBLP Computer Science bibliography. https://dblp.org
[2] [n. d.]. The Heartbleed Bug. https://heartbleed.com/
[3] [n. d.]. Peach fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/
[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/
[5] Frederic Besler. 2016. Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/
[6] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Softw.* 38, 3 (2021), 79–86. https://doi.org/10.1109/MS.2020.3016773
[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020
[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[9] Hanno Böck. [n. d.]. How Heartbleed could've been found. https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html
[10] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 181–195. https://doi.org/10.1145/3192366.3192388
[11] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, Carey Williamson, Aditya Akella, and Nina Taft (Eds.). ACM, 475–488. https://doi.org/10.1145/2663716.2663755
[12] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi
[13] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. 2019. AntiFuzz: Impeding Fuzzing Audits of Binary Executables. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1931–1947. https://www.usenix.org/conference/usenixsecurity19/presentation/guler
[14] Sam Hocevar. [n. d.]. zzuf. https://github.com/samhocevar/zzuf
[15] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. 474–484. https://doi.org/10.1109/ICSE.2012.6227168
[16] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*,

Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1913–1930. https://www.usenix.org/conference/usenixsecurity19/presentation/jung

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[18] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2777–2794. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[19] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[20] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/2491411.2491438

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

[22] Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. 2022. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Trans. Softw. Eng.* 48, 6 (jun 2022), 2028–2039. https://doi.org/10.1109/TSE.2020.3047766

[23] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 697–710. https://doi.org/10.1109/SP.2018.00056

[24] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Trans. Software Eng.* 47, 9 (2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

[25] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[26] Eric F. Rizzi, Sebastian G. Elbaum, and Matthew B. Dwyer. 2016. On the techniques we create, the tools we build, and their misalignments: a study of KLEE. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 132–143. https://doi.org/10.1145/2884781.2884835

[27] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[28] Kostya Serebryany. 2016. Sanitize, Fuzz, and Harden Your C++ Code. USENIX Association, San Francisco, CA.

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

[30] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 138–157. https://doi.org/10.1109/SP.2016.17

[32] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1275–1295. https://doi.org/10.1109/SP.2019.00010

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

[34] Michal Zalewski. [n. d.]. AFL Historical Notes. https://lcamtuf.coredump.cx/afl/historical_notes.txt

[35] Michal Zalewski. [n. d.]. AFL README. https://lcamtuf.coredump.cx/afl/README.txt

[36] Michal Zalewski. [n. d.]. AFL Website. https://lcamtuf.coredump.cx/afl/

[37] Marcel Zalewski and Marcel Böhme. [n. d.]. AFL mailing list discussion of AFLFast. https://groups.google.com/g/afl-users/c/fOPeb62FZUg