On the Influence of Compiler Optimizations on Fuzzing

MATTHEW WEINGARTEN, Automated Software Testing, ETH Zurich, Switzerland
ANDRIN BERTSCHI, Automated Software Testing, ETH Zurich, Switzerland

The automated software testing technique fuzzing has seen a golden age in the last decade, with widespread use in industry and academia. On the hunt to find vulnerabilities, fuzzing binaries are compiled with default compiler optimizations such as -O2, or -O3, which remain the hard-coded default in popular fuzzers such as AFL++. On a binary level, software compiled from the same source code may vastly differ in control flow depending on used compilation flags. In this work, we aim to analyze the impact of different compiler optimizations on the fuzzing process and provide further insight. We influence compilation passes of the clang/LLVM compiler and analyze their impact on the fuzzing performance of AFL++. We integrate our work into Fuzzbench, an open-source fuzzing pipeline, and run experiments on real-world benchmarks. Our preliminary fuzzing results indicate that there is a delicate trade-off between runtime performance and code complexity. While our results show significant differences on the scale of individual benchmarks, when summarizing across the whole bench suite, there is no evidence to suggest a statistical difference in fuzzing performance.

1 INTRODUCTION

Fuzzing has become an integral element in the hunt to find software vulnerabilities and bugs. It is continuously used to discover bugs in large and real-world code bases like operating systems, compilers, and web browsers [clu [n.d.]; Miller et al. 2020, 2006]. Before starting a fuzzing campaign, one must compile the project with a toolchain adding instrumentation code to the Unit Under Test (UUT). A coverage-guided fuzzer relies on this instrumentation to receive feedback for generated test cases, such that it can maximize exploration of "interesting" code areas.

Much research has gone into good seed and input generation strategies [Aschermann et al. 2019; Chen et al. 2018; Gan et al. 2018; Rawat et al. 2017], but there is no comprehensive analysis of how the plethora of compiler options affect fuzzing performance. The goal of standard compiler optimizations and fuzzing are inherently misaligned; while compilers generally optimize for execution speed and code size, a fuzzer's goal is to detect faulty code. This misalignment may manifest itself in an unnecessary loss in fuzzing performance based on the chosen compiler flags. Practitioners typically fuzz binaries compiled with default compiler optimizations such as -O2, or -O3, and these flags remain the default in many fuzzers such as AFL++ [Fioraldi et al. 2020]. To the best of our knowledge, no research has been done on the influence of standard compiler flags on fuzzing performance.

AFL++ is a popular coverage-guided fuzzer and subject to much research interest and continuous improvements from academia and private industry[1]. The LLVM project [Lattner and Adve 2004] is a suite of reusable compiler technologies and tools and the backbone of many industry-leading technologies such as clang[2].

[1]Research with AFL++ https://aflplus.plus/papers/
[2]Clang the C front-end for LLVM https://clang.llvm.org/

Authors' addresses: Matthew Weingarten, matthew.weingarten@inf.ethz.ch, Automated Software Testing, ETH Zurich, Switzerland; Andrin Bertschi, andrin.bertschi@inf.ethz.ch, Automated Software Testing, ETH Zurich, Switzerland.

In this work, we aim to analyze the effects of clang/LLVM compiler flags on fuzzing performance. We modify AST++ to influence compilation flags and extend Fuzzbench [Fuzzbench 2020], an existing fuzzing pipeline, to systematically benchmark, analyze and plot results.

(1) How do compiler flags affect fuzzing performance?
(2) Can we find a set of flags and achieve an improvement over standard 0-level flags?
(3) How much performance gain can we get from combining multiple fuzzing runs on different output binaries?

Our fuzzing results on real-world software indicate there is no significant impact of standard compiler flags on fuzzing performance. No set of flags outperforms other flags across a set of benchmarks. While our results are constrained by limited fuzzing compute, we observe there is a delicate trade-off between runtime performance (faster fuzzing allows for more runs), and code complexity (more edges give more feedback). We hope to lay the foundation for more work on the influence of compiler optimizations on fuzzing.

2 BACKGROUND

Defining Fuzzing Performance. The intention of fuzzers to find bugs and vulnerabilities does not lend itself naturally to evaluating the performance of fuzzers. Large-scale real-world software projects typically contain a sparse number of bugs in relation to code size. Bugs have to be reported through crash reports, leading to an additional layer of indirection. This mapping from crashes to bugs is evaluated differently in literature and may also cause bug number inflation [Klees et al. 2018]. Good fuzzing benchmarks also add synthetic bugs to their test suite [Fuzzbench 2020; Hazimeh et al. 2020]. Combined, these issues may cause reported performance results to be misleading, especially when running with limited CPU hours.

These facts suggest using coverage metrics as a performance criterion. However, previous research has found that there is only a weak correlation between any form of coverage metric and finding bugs[Inozemtseva and Holmes 2014; Klees et al. 2018]. These conclusions should be taken into account when evaluating performance data, especially when making statements about real-world fuzzer performance implied by coverage metrics. Nevertheless, in this paper, we only consider edge coverage, described in more detail in Section 3. Ideally, any claims on fuzzing performance should be made on bug finding ability, even if one follows the adage [Aschermann et al. 2019] of

No fuzzer can find bugs in code that is not covered.

Still, this remains a limitation of our analysis and is left as future work.

Effects of Compiler Flags on Fuzzing. Coverage-guided grey box fuzzing uses instrumentation to record which edges have been covered so far. Typically, when a test case triggers a new path it is stored by AFL++. The new test case is subsequently mutated to hit

```
int foo(int x, int y, int z, int w){
int result = 0;
 if (x == y) {
  z = x + y;
  if (w == x) {
  /* Bad code */
   result = x + y + z;
  }
 }
 return result;
}
```

```
void foo(int x, int y, int z, int w){
 int result = (x + y) << 1;
 if (x == y && x == w){
  /* Bad code */
  return result;
 } else {
  return 0;
 }
}
```

Fig. 1. Background to motivate the effect of compiler flags on fuzzing: A simplified branch elimination optimization, see Figure 2 and Figure 3 for their low-level representation. The code on the left is reconstructed from LLVM after O3 has been applied, while the left is the source.

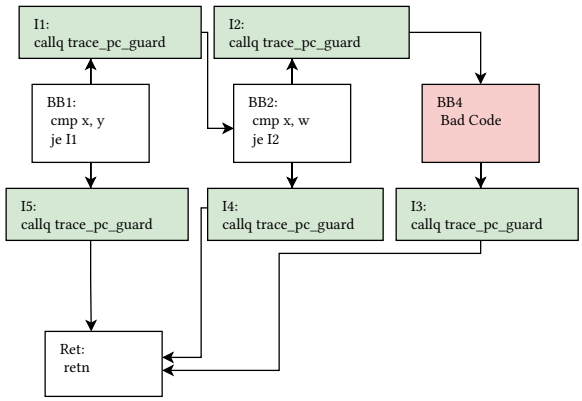

Fig. 2. Background to motivate the effect of compiler flags on fuzzing: Illustrates control flow graph of non-optimized code. In green, we see coverage instrumentation, in red we see the faulty code. Note how a fuzzer has to overcome simpler branches to reach the instrumentation code, compared to the optimized code in Figure 3. Only control flow statements are preserved, the rest is omitted for legibility.

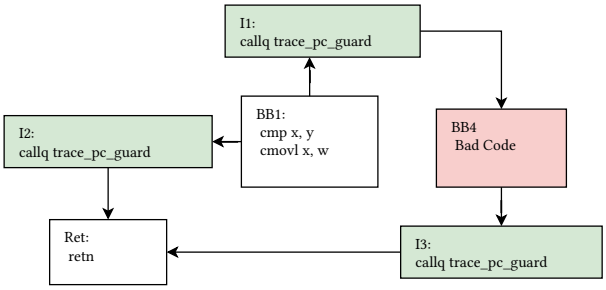

Fig. 3. Background to motivate the effect of compiler flags on fuzzing: Illustrates control flow graph of optimized code. The compiler eliminated a basic block and merged two conditionals into one. Only control flow statements are preserved, the rest is omitted for legibility[3].

previously undiscovered instrumentation points, and the cycle repeats. Usually, each edge in the CFG is instrumented. The feedback loop of fuzzing is thus heavily affected by the structure of the CFG. In Figures 1, 2, 3, we motivate how standard compiler optimizations and fuzzing are inherently misaligned; while compilers generally optimize for execution speed and code size, a fuzzer's goal is to detect faulty code. Figure 2 illustrates a simplified view of the CFG created by source in Figure 1 without optimizations. The fuzzer can receive incremental feedback for mutated inputs. If an input is generated causing parameters $x == y$, a new edge is reached and the new input is added to the corpus. Now an upcoming input can be generated by the previous input, and only needs to fulfill $x == w$. Now, in contrast, Figure 3 shows the same source code compiled with O3. This time, to reach the "Bad code", the fuzzer must generate an input to fulfill the condition $x == y$ *and* $x == w$. This means if an input is generated such that $x == y$, but $w! = x$, the input is discarded and no longer added to the corpus. In a nutshell, the probability of a fuzzer reaching the "Bad code" region is higher with flags O0 than with flags O3.

3 APPROACH

We forked Fuzzbench [Fuzzbench 2020] and extended it with additional fuzzers to suit our needs. This way we can use Fuzzbench as a fuzzing-as-a-service model. Critically, Fuzzbench already provides plumbing code to clone, build and instrument well-fuzzed (OSS-Fuzz) benchmarks, with default input seeds. It is, however, optimized for the cloud[4]. Fuzzbench employs the *LLVM Source-based Code Coverage*[LLVM 2022] to compare and evaluate fuzzing performance across different fuzzers. While source-based code coverage is a suitable metric to compare different fuzzers with another, it is too coarse-grained for our work. We are interested in the control flow changes across different compiler flags for the same fuzzer. This is why we implemented different coverage instrumentation for Fuzzbench, which we further detail in Section 3. The high-level fuzzing pipeline is depicted in Figure 4. We now describe the high-level pipeline and then proceed to document implementation details.

Fuzzing Pipeline. Our fuzzing pipeline is based on Fuzzbench [Fuzzbench 2020] with additional fuzzers and a custom coverage

[4]For each fuzzing execution, a docker image is built which causes overhead when run on a local machine.

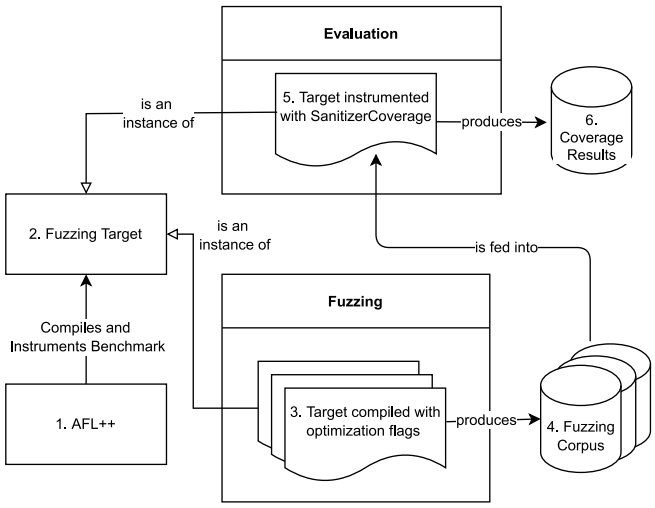

Fig. 4. Big Picture of Fuzzing Pipeline. Steps 1. - 6. indicate the pipeline steps to instrument, fuzz, and evaluate the impact of different optimization flags on the overall fuzzing performance.

evaluation. In order to fuzz a fuzzing target, we use AFL++ (1. in Figure 4) on a set of well-studied fuzzing targets. We compile and instrument the targets with AFL++'s coverage-guided instrumentation and produce several binaries (2.). These binaries differ in the set of optimization flags passed to either clang or LLVM optimizer. We subsequently fuzz these binaries (3.) for a predefined time and produce fuzzing corpora (4.). Fuzzbench creates a snapshot of the obtained corpora in a configured interval of 15 minutes. Each snapshot of the corpus is then fed in the evaluation phase into a reference target (6.) compiled with *No optimizations* (-O0). This reference binary contains LLVM SanitizerCoverage instrumentation on each edge in the control flow graph of the *-O0* compiled target. During the evaluation, we feed the corpora obtained from different optimization flags into the reference binary and obtain coverage results (6).

AFL++ Instrumentation. We use AFL++ as our fuzzer with the clang/LLVM backend. We modified AFL++ to accept additional compiler flags for the instrumentation. The compiler flags can be set as an environment variable as illustrated below.

```
AST_CC_ARGS="-O3 -fno-unroll-loops -fno-vectorize" \
AFL_DONT_OPTIMIZE=1 afl-cc <binary>
```

We try to follow a realistic AFL++ configuration and use a dictionary, *persistent mode*, *Fast LLVM-based instrumentation (afl-clang-fast)*, and *CmpLog* to overcome checksums and magic bytes. These configurations were used in [Fuzzbench 2020] as a baseline for an AFL++-based fuzzer.

LLVM Optimizer. The clang compiler does not expose the underlying LLVM opt passes but rather provides its own set of compiler flags to guide the compilation process. This is an explicit design decision by the developers in order not to expose the optimization pipeline details to consumers [llv [n.d.]]. To obtain more fine-grained control, we propose the following compilation steps.

```
#!/usr/bin/env bash
CLANG_FLAGS="-flto -O1 -Xclang -disable-llvm-passes"
OPT_FLAGS="--simplifycfg" # opt flags to expose

afl-clang-lto "$@" $CLANG_FLAGS
for arg in "$@"
do
    if [[ "$arg" == *.o ]]
    then
        opt $OPT_FLAGS $arg -o $arg
    fi
done
```

Listing 1. Compiler wrapper script to influence LLVM opt when compiling fuzzing target. LLVM opt passes are not exposed to clang by design. This snippet is an attempt to influence LLVM opt without modifying the build scripts of the fuzzing target. The example above will run the simplifycfg pass on each bitcode file

(1) We expose a clang wrapper which compiles the fuzzing target with `-disable-llvm-passes` to prevent any automatic LLVM opt optimization passes,
(2) We compile using Load-Time-Optimizations (-flto). This instructs clang to emit object files containing LLVM bit code instead of machine code output,
(3) After compilation of each object file, we parse the command line arguments and feed each object file into LLVM opt, overwriting the original object file with its optimized bitcode,
(4) The object files are then linked as usual in the linking phase of compilation. Due to `-disable-llvm-passes`, the linker will not further optimize the program.

This idea is further illustrated with the snippet in Listing 1.

Clang implements LTO by handling LLVM bitcode files as object files. This postpones whole program optimization into the linking phase and allows the linker to run LLVM optimization passes on the entire program bitcode before converting it to a native executable[5]

While this approach lets us to influence the LLVM optimizer, the optimization passes are limited to the scope of an object file. Additionally, it requires the LTO version of AFL++. Further work may implement a custom LLVM pass manager for clang and directly controls optimizer passes. A custom pass manager, however, was outside of the time scope of our work. Our current approach allows us not to modify the build systems of the target binaries and hence easily scales to a wide set of benchmarks.

Benchmark Instrumentation. Recall from Figure 4 that we feed the fuzzing corpora into a reference binary to obtain coverage results. The reference binary is compiled using *No optimization*, e.g. `-O0`. As already identified in Section 3, we replaced Fuzzbench's Source-based coverage with SanitizierCoverage[6]. We instrument the target binary with `trace-pc-guard` and `no-prune` and use the default implementation of sanitizer runtime to trace the control flow on all edges in the target binary. During the evaluation, we feed the fuzzing corpora into the reference binary and extract all covered edges as

[5]https://llvm.org/docs/LinkTimeOptimization.html
[6]https://clang.llvm.org/docs/SanitizerCoverage.html

```
extern "C" int LLVMFuzzerTestOneInput(
        const uint8_t *Data, size_t Size)
{ /* Do something */ }
```

Listing 2. Libfuzzer entry point for fuzzing targets. All Fuzzbench fuzzing targets already include this endpoint.

a SanitizierCoverage *sancov* dump. In subsequent post-processing, we analyze and compare the covered edges.

All Fuzzbench benchmarks[7] use libFuzzer[8] to implement a fuzz target. The libFuzzer project suggests an entry point for a fuzzing engine to fuzz a target. This entry point accepts an array of bytes and does something interesting with these bytes during fuzzing (listing 2). The entry point is a weak symbol and does not require the libFuzzer runtime.

Libfuzzer deprecated support to dump *sancov* coverage when compiled with `trace-pc-guard`. Our coverage evaluation depends on the *sancov* dump feature of SanitizierCoverage which is why we link without the libFuzzer runtime and use a simple test driver to feed the corpus into the target binary.

Choosing Optimization Flags. Benchmarking the performance of a fuzzer requires a large amount of CPU time. The authors of Fuzzbench claim that it can take up to 11 CPU years to run a well-conducted Fuzzbench experiment [Fuzzbench 2020]. Hence, running every compiler flag is infeasible in a reasonable time. Even worse, compiler phases interactively enable one another, theoretically requiring testing of every possible flag combination for an exhaustive analysis. We hand-pick a small subset of flags to reduce the search space.

Choosing the correct flags to analyze experimentally implies first weighing the cost-benefit potential of each flag and their interactions in theory. We believe there is a delicate trade-off here: The faster a binary runs, the higher the throughput of testing mutated inputs is. On the other hand, optimizations may hinder the likelihood of a single input mutation to find a new coverage area [Simon and Verma 2020].

Our philosophy was to scan for phases that make the most dramatic changes to the CFG. For example, it is hard to justify how phases such as *Constant Propagation* or *Global Value Numbering* could in any way affect fuzzing performance unless they open up opportunities for CFG-modifying phases. In contrast, the phase *Simplify CFG* can merge basic blocks, potentially changing the coverage feedback AFL++ can receive.

The main flags exposed to developers are `-O` levels. To analyze the differences between these flags, we focus on the hidden options passed to the LLVM optimizer. Refer to Table 1 for a collection of options we have chosen to observe more closely. The table is not an exhaustive list of all differences in -O levels. There are around 80 additional passes in O1 compared to 00, 15 in O1 to O2 and 3 in O2 to O3 (all of which are listed). Before conducting experiments we suspected some passes to potentially harm fuzzing, as show in the last column. We believe *simplifycfg* may harm performance by merging basic blocks and thus conditions (similar to example in Figure 1). Both *loop-unroll* and *loop-simplify* may cause structural changes in the CFG. Vectorizing may cause conditional statements to be masked on an entire vector of values. A fuzzer must meet all these values with no feedback along the way.*constmerge* and *argpromotion* are 2 of the 5 -O3 specific flags []. We do not suspect them of harming fuzzing, as they do not modify the CFG [pas [n.d.]]. Interestingly, *callsite-splitting* may even help fuzzing performance, as it adds additional branches to the CFG [cal [n.d.]]. Finally, *aggressive-instr* aims to merge conditional statements [pas [n.d.]], and, as shown in Section 2 causes trouble.

[7]Note we use target binary and benchmark interchangeably in this document
[8]https://llvm.org/docs/LibFuzzer.html

LLVM opt passes	O0	O1	O2	O3	Potentially harmful
simplifycfg	No	Yes	Yes	Yes	Yes
loop-simplify	No	Yes	Yes	Yes	Yes
loop-unroll	No	Yes	Yes	Yes	Yes
loop-vectorize	No	Yes	Yes	Yes	Yes
slp-vectorizer	No	No	Yes	Yes	Yes
callsite-splitting	No	No	Yes	Yes	No
constmerge	No	No	No	Yes	No
argpromotion	No	No	No	Yes	No
aggr-instcombine	No	No	No	Yes	Yes

Table 1. Table of an excerpt of LLVM opt compiler flags and their occurrence in the standard optimization levels O0 to O3.

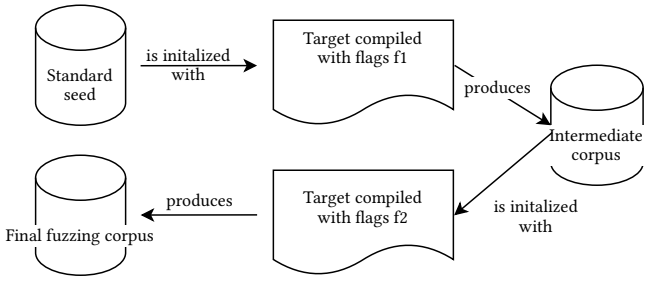

Fig. 5. Combining different fuzzing runs with distinct compilation flags. A fuzzing target is compiled with flags *f1*, and *f2*, the intermediate corpus of *f1* is fed into *f2*.

According to our thought process, there are only a handful of passes that actively harm performance, while the rest either do not affect, or even improve fuzzing performance by providing execution speedups. This motivated our *aflplusplus_lto_l1* configuration, seen in Figure 14.

Combining Different Fuzzing Runs. We observed that fuzzers often reach a point of saturation, where little to no progress is made once a coverage threshold is reached [Fuzzbench 2020; Hazimeh et al. 2020; Li et al. 2021]. Instead of running the fuzzer for a time period with one set of compiler flags, breaking down the fuzzing run into multiple steps with different binaries may be a remedy to help push the saturation point back.

We hypothesize why this might be the case: let us assume we have an arbitrary instrumented edge e in a binary compiled with flag $f1$. Due to the nature of fuzzing and its input generation, there is a likelihood $p1$ such that the fuzzer reaches e in the available runtime. Once the coverage threshold for high probability branches is reached, much time is spent trying to reach unlikely edges. Now, assume we compile the code with another flag $f2$, and assume all the previous knowledge gained from $f1$ has persisted, the probability profile of an edge might change, giving us $p2$ for e. If $p2 > p1$, there is a higher likelihood for the fuzzer to now discover e. If the probability distribution of all edges is different enough, this could increase the final coverage.

How would the probabilities $p1$ and $p2$ change for different flags? Our reasoning is twofold. First, the probabilities may change solely based on the structure of the CFG and changes in the IR, as discussed in Section 2. On top of this, the probabilities could be affected by hash collisions in the AFL fuzzer itself. The main downside of this approach is losing all AFL++ metadata across runs, for example, if the state of the input generation strategy has been changed or certain heuristics are used.

To evaluate our hypothesis, we split the run of 4 hours into two 2-hour blocks, as depicted in Figure 5. In this case, the final corpora of a fuzzing run with flag $f1$ is fed into the run with $f2$ as a seed input. We then evaluate the final corpora and compare it with other compilation flags. To choose two flags $f1$ and $f2$ to combine, we want to find the flags with the most disjoint edge reaching probability distribution. This can be approximated by analyzing the pairwise unique edge coverage and choosing configurations with the highest overall difference, see Figure 12 for the pairwise unique edge coverage.

Due to limited budget, we evaluate on a combined run, namely O2 and O3. We chose two easily accessible top-level flags that had the most unique edges covered, see Figure 12.

4 IMPLEMENTATION AND RESULTS

In the upcoming section we evaluate and compare the performance of different compilation flags on the fuzzing process. We describe the different compilation flags, introduce the benchmarks and experimental setup, and present the preliminary fuzzing results.

4.1 Implementation

Optimization Flags. Figure 14 enumerates all optimization flags. We experiment with the default optimization levels O0 to O3 as well as emit platform specific (-march=native) assembly. *Aflplusplus_ast_f0* and *aflplusplus_ast_f1* contain O3 optimizations excluding loop unrolling and function inlining. We believe that these passes may remove interesting edges which otherwise allow AFL++ to obtain more coverage feedback.

As we have established with Listing 1, some optimizations require afl-clang-lto. To obtain better baselines, we include a few fuzzing experiments including load-time-optimizations (LTO) such as *aflplusplus_ast_opt_disable* and *aflplusplus_ast_o3_lto*. The compiler flag bundle *aflplusplus_ast_l1* keeps all O3 flags but removes flags we suspected to be harmful to fuzzing performance.

```
loop-unroll, loop-simplify,
loop-vectorize, slp-vectorizer,
aggressive-instcombine, simplifycfg
```

Lastly, *aflplusplus_ast_combined_o3_o2* implements the combination of different optimization flags and runs as introduced in Figure 5.

Fuzzing Target	Format	Edges	Seeds
vorbis-2017-12-11	OGG	5022	1
freetype2-2017	TTF, OTF, WOFF	19056	2
libjpeg-turbo-07-2017	JPEG	9586	1
harfbuzz-1.3.2	TFF, OFT, TTC	10021	58
bloaty_fuzz_target	ELF, DWARF, MachO	89530	94
lcms-2017-03-21	ICC profile	6959	1
libpcap_fuzz_both	PCAP	8149	0
openthread-2019-12-23	custom	17932	0
sqlite3_ossfuzz	custom	45136	1258
woff2-2016-05-06	WOFF	10923	62

Table 2. Fuzzbench targets used for experiments.

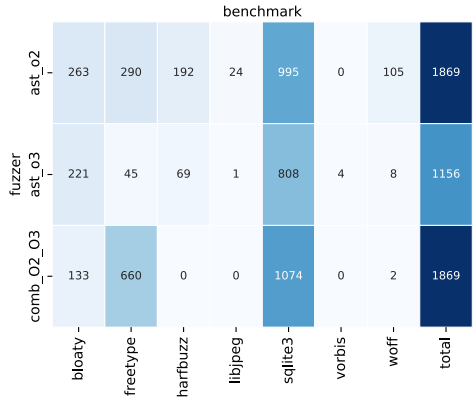

Fig. 6. Number of unique edges covered that are not covered by any other benchmark. This was plotted separately to visualize the data in isolation between the combined run, and the two configurations making up the combination.

Fuzzing Targets. In Figure 14, we summarize the benchmarks used in our the experiments. The benchmarks are part of the Fuzzbench platform and provide a wide range of input formats.

Experimental Setup. For each benchmark/flag pair we run 3 trials for at least 4 hours, as less fuzzing time leads to high variance in fuzzing performance. However, due to budget constraints, we execute the benchmarks as part of the Fuzzbench pipeline within dedicated docker containers on a local machine and not in the cloud.

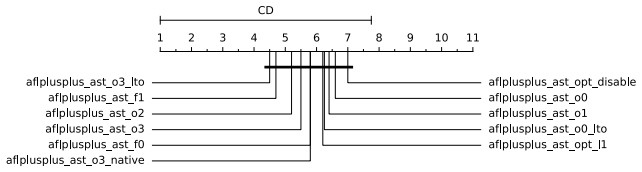

Fig. 7. Critical Difference Plot according to Demsar et al. [Demšar 2006], compares all benchmarks results with each other according to their average rank (in terms of edge coverage) and the statistical significance between them. Average rank is calculated based on medians of coverage on each benchmark, then averaged across all benchmarks. Fuzzers connected by the bold line indicate that there is no significant difference between them.

In snapshot intervals of 15 minutes we copy the current corpus and feed it into the evaluation pipeline to obtain coverage results.

Hardware Configuration. We run experiments on the hardware listed below.

Laptop 1 Ubuntu 20.04.4 LTS, 16 core AMD(R) Ryzen 7 pro 4750u CPU @ 3.6GHZ, with 32 GB of RAM.

Laptop 2 Fedora 35, Kernel 5.17, 6 core Intel(R) Xeon E-2176M CPU @ 2.7GHz, with 16GB of RAM.

Bloaty_fuzz_target, lcms-2017-03-21, openthread-2019-12-2, sqlite3 ossfuzz, woff2-2016-05-06 are benchmarked on Laptop 1, while freetype2-2017, libjpeg-turbo-07-2017, harfbuzz-1.3.2 and vorbis-2017-12-11 are benchmarked on Laptop 2.

4.2 Results

In this section we present our final results of the fuzzing campaign.

Critical Difference. With Figure 7 we establish a *Critical Difference* according to the definition of Demsar et al. [Demšar 2006]. The critical difference plot compares all compiler flags on all benchmarks with each other according to their average rank (edge coverage) and the statistical significance between them. The statistical significance is computed using a post-hoc Nemenyi test performed after the Friedman test [Demšar 2006; Fuzzbench 2020]. According to this definition, the fuzzing results show no significant difference between the tested compiler flags and fuzzing performance. However, we want to emphasize that the fuzzing campaign was constraint in resources.

No One-size-fits-all. Figures 8 and 10 present box plots of different compiler flags on two benchmarks, woff2-2016-05-06 and sqlite3_ossfuzz respectively. More box plots are attached in the appendix in Section 7. Interestingly, while the LTO versions of the fuzzers perform worse than the rest in Figure 8, they perform better than all the rest in Figure 10. This suggests that there is no *One-size-fits-all* configuration across all benchmarks.

Figures 9 and 11 show two excerpt of an Vargha-Delaney A12 plot of the aforementioned two benchmarks [Vargha and Delaney 2000]. The plot emphasizes on average, how much better one pair of flags is than an other. The measure ranges from 0 to 1. A score of 0.5 implies that both compiler flags achieved equal performance. The plots emphasize what the previous box plots already showed:

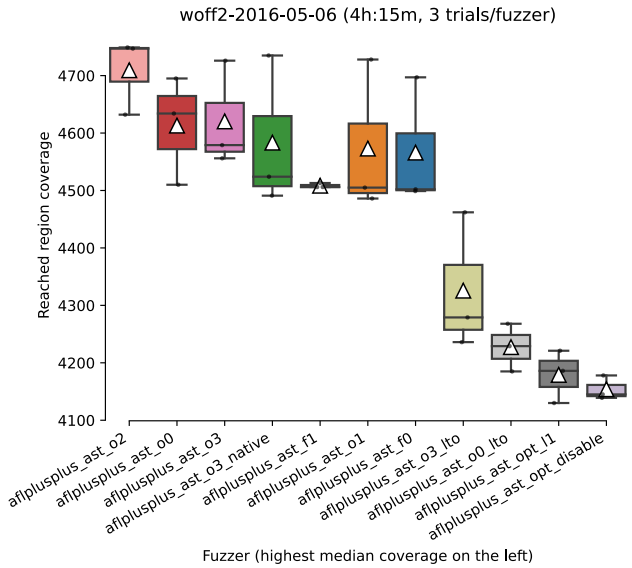

Fig. 8. Boxplot for woff2-2016-05-06 fuzzing target.

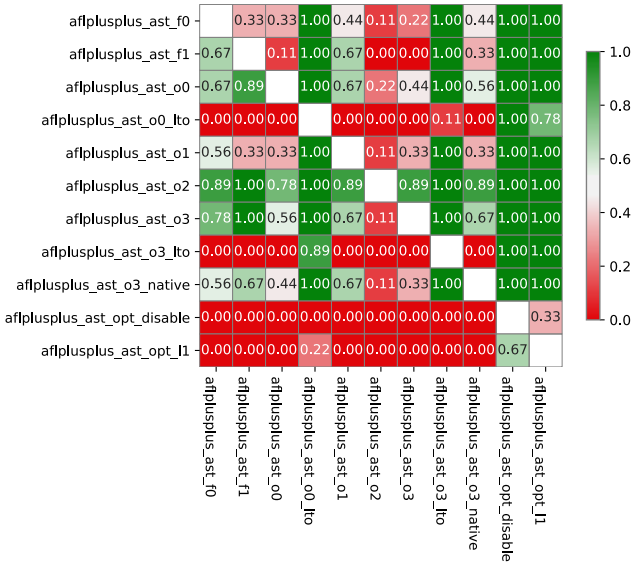

Fig. 9. Varga delaney plot for woff2-2016-05-06 fuzzing target. The plot is to read in the following way: Fuzzer in the row outperforms fuzzer in the column on average with the given probability.

in some benchmarks, certain flags outperform others, however not consistently across benchmarks.

Differences in covered Edges. The number of edges a fuzzer reaches is a quantitative measure of performance, but does not necessarily give the complete qualitative picture. While there may be no statistical difference between two fuzzers, a fuzzer still performs better if it can consistently reach edges another cannot. Figure 12 shows

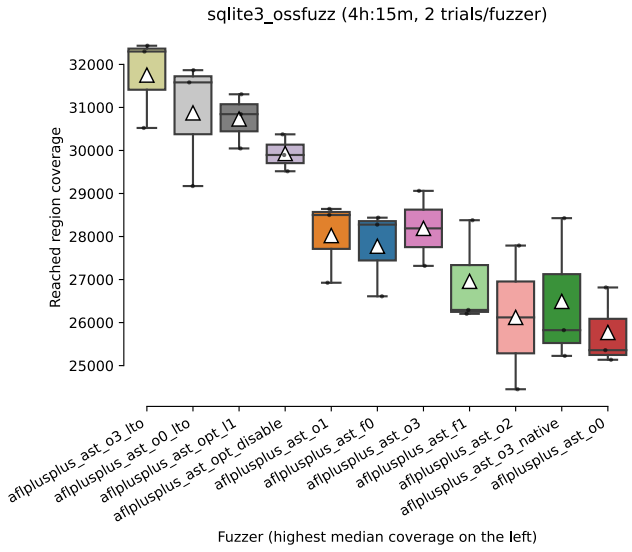

Fig. 10. Boxplot for sqlite3_ossfuzz fuzzing target.

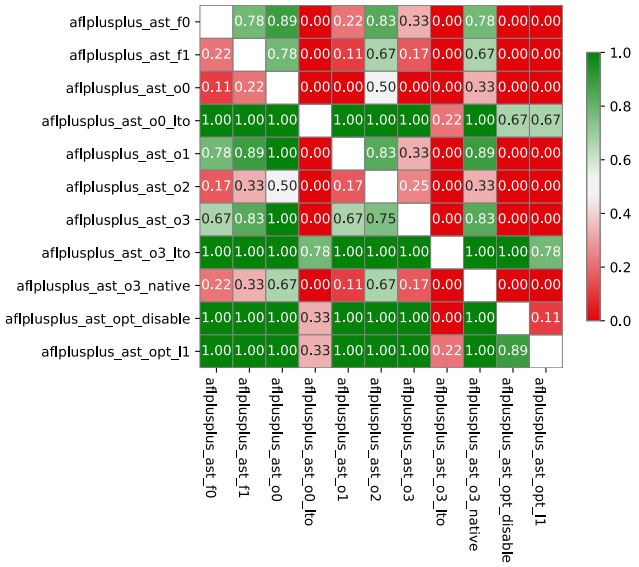

Fig. 11. Varga delaney plot for sqlite3_ossfuzz fuzzing target.

pairwise comparison for each fuzzer. The values are normalized for each benchmark between 0 and 1 and aggregated by mean across all benchmarks. Normalization is required, as the distinct edges on benchmarks range from 1 - 2 to many thousand. The set of edges reached for each fuzzer is the union of all reached edges in all trials.

For example *ast_o2* covers a high number of edges that *ast_o0_lto* doesn't (value 0.5), but vice versa *o0_lto* covers many that *O2* doesn't. This implies the probability distribution for reaching edges is more disjoint than *f1* and *f2*.

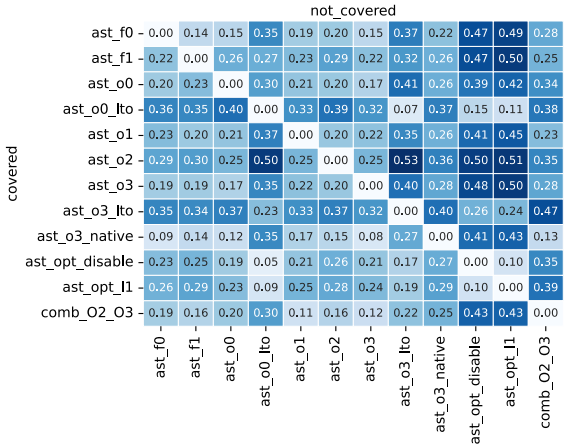

Fig. 12. quantifies the number of edges covered by left fuzzer which are not covered by bottom fuzzer. The values are normalized for each benchmark between 0 and 1 and subsequently aggregated across all benchmarks. The higher, the more unique edges are covered.

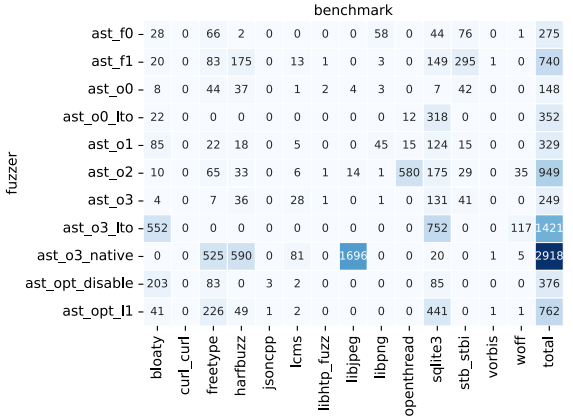

Fig. 13. shows the number of edges covered by a fuzzer that are not covered by *any* other fuzzer for each benchmark. The higher, the better a fuzzer performs.

Complementary to Figure 12, Figure 13 shows for every fuzzer and every benchmark the number of flags *no other fuzzer reached*. This provides additional insight into fuzzing performance. Interestingly, *ast_o3_native* did not find significantly more edges than any other fuzzer in the benchmark libjpeg (even did mostly worse, see Figure 15), but found 1696 edges that no other fuzzer found. Further analysis is required to figure out if this happens consistently, or is a random anomaly, or caused by short fuzzing times, but understanding the root cause of these results could lead to future increases in fuzzing performance.

Results in Combined Flag. The combined flags yielded interesting results. Figure 6 shows that the combined flags found more unique edges than O2 or O3 in sqlite3 and freetype, but less in the others. Overall, it did consistently outperform any other fuzzer configuration in the complete picture Figure 15. We refrain from making sweeping statements regarding the combined flags, as we only have a limited time budget to run and are only able to compare with a subset of benchmarks. Additionally, we lack statistical tests for the combined flags.

Results for ast_lto_l1. The flags were hand-picked by us attempting to achieve the highest possible fuzzing performance by only changing compiler flags. Our hypothesis and reasoning in previous sections did not manifest themselves in empirical evidence to support our claim. However, longer fuzzing trials are required before discarding the hypothesis.

5 RELATED WORK

Controlled Compilation. The work in 'Controlled Compilation' studies how standard compiler toolchains undermine fuzzing performance [Simon and Verma 2020]. A new toolchain was built on top of Clang/LLVM by removing passes that harm fuzzing and adding additional passes to aid fuzzers. Here we focus mostly on their discoveries of how flags hurt fuzzers.

They identify that the major types of optimizations harming performance are ones merging conditional statements and eliminating branches. They describe the pass *simplifycfg* and how it can merge conditions and reduce fuzzing feedback. They also show how branch-less optimizations eliminate edges, also harming fuzzing performance. Similarly, they name vectorization has the same issue, where conditionals are often performed on many values simultaneously, again merging conditional statements. All of these passes make a mutation less likely to hit a new instrumentation point, and thus harm fuzzing performance.

The author's initial instinct was to disable all optimization, but this technique proved unsatisfactory. They reported minimal to no difference between fuzzers with `O0` and `O3`. They report an increase in the number of bugs found using AFL from 6 total bugs to up to 27 bugs on a benchmark-suite of 6 benchmarks sampled from binutils, libpng, and libxml among others. However, they do not continue to provide experimental evidence of differences in fuzzing performance between standard compiler flags, and completely disregard levels `O1` and `O2`.

LAF-Intel. This work goes through a single example of libpng, where an LLVM pass undoes previous passes that merge conditions and propose splitting difficult to fulfill conditions in smaller chunks for better fuzzing feedback [laf [n.d.]]. They report an increase of 23% in lines covered after 52 hours of fuzzing. They essentially undo harmful passes that merge conditional statements. However, this approach may be difficult to scale, as it may require a lot of manual effort by rewriting undo transformations.

Instruguard. Errors in the instrumentation of the binary can result in performance degradation. Instruguard [Liu et al. 2021] is a tool for analyzing the source code produced by a fuzzing wrapper compiler (like afl-clang or afl-gcc), detecting instrumentation errors, and resolving them. Instrumentation errors include missed and redundant instrumentation locations, where the missed location may cause fuzzers to discard test cases even when interesting and redundant locations can mislead fuzzers with faulty feedback and exacerbate hashtable collisions.

Phase	**instrumentation errors caused**
-simplifycfg	29.6 %
-jump-threading	26.2 %
-inline	13.2 %
-loop-rotate	6.2 %
-other	34.8 %

Table 3. Related Work: Distribution of instrumentation errors root causes reported in [Liu et al. 2021]. Passes explicitly mentioned without exact percentage as part of *other* include *-loop-unroll*, *-loop-simplify*

They measured that over 50% of all basic blocks compiled with AFL++ contain an instrumentation error. This high number of errors raises the question of how much of the performance difference with different flags are due to an indirect effect of instrumentation errors, or the difference in feedback itself. Fixing the instrumentation code, they reported an overall coverage increase for AFL from 33.4% to 33.7% and a bug finding increase from 19 to 22. Unfortunately, they do not show an increase in performance for AFL++.

See table 3 for the percentage breakdown of compiler flag root causes for instrumentation errors. Interestingly, there are four different flags that overlap with flags we deemed interesting in section 3,namely *simplifycfg*, *inline*, *loop-unroll* and *loop-simplify* and are among the main contributors to instrumentation errors. Nevertheless, the preliminary data shows this may be a promising area for future research, especially considering how disjoint the number of edges detected are in Figure 12.

6 CONCLUSION

In this work we presented the analysis of clang/LLVM compiler flags on fuzzing performance of AFL++, a popular coverage-guided fuzzer. We integrated our analysis into Fuzzbench, a fuzzing pipeline, to systematically benchmark, analyze, and plot our findings. We benchmarked a set of 8 real-world software projects during a period of 4 hours and 3 trials.

(1) How do compiler flags affect fuzzing performance? No statistical difference was found in the number of edges covered across all benchmarks, regardless of compiler flag configuration. Even so, we observe a range of unique edges covered, either pairwise or across all configurations. This suggests that compiler flags will affect which edges get covered and which do not. However, these results need to be further analyzed with longer fuzzing times and require inspection of how consistently the same results are achieved. We attempted to leverage this to get better performance by combining flags, for the research question (3).

Fuzzer	Compiler Flag	Description
aflplusplus_ast_o0	*-O0*	No Optimizations
aflplusplus_ast_o1	*-O1*	O1 Optimizations
aflplusplus_ast_o2	*-O2*	O2 Optimizations
aflplusplus_ast_o3	*-O3*	O3 Optimizations
aflplusplus_ast_o3_native	*-O3 -march=native*	O3 with CPU specific instructions
aflplusplus_ast_f0	*-O3 -fno-unroll-loops -fno-vectorize -fno-slp-vectorize -fno-inline-functions*	O3 without loop unrolling and function inlining
aflplusplus_ast_f1	*-O3 -fno-unroll-loops -fno-vectorize -fno-slp-vectorize*	O3 without loop unrolling
aflplusplus_ast_opt_disable	*afl-clang-lto with -flto, -O1, -Xclang, -disable-llvm-passes*	LTO with all optimizations disabled
aflplusplus_ast_o3_lto	*afl-clang-lto with -flto, -O3*	LTO with O3
aflplusplus_ast_opt_l1	*afl-clang-lto with opt flags*	O3 and harmful flags removed
aflplusplus_ast_combined_o3_o2	*combined*	Run O3 for 2h , then run O2 2h

Fig. 14. Experiment names and their used compilation flags. If not otherwise stated, we fuzzed with `afl-clang-fast/++`.

(2) Can we find a set of flags and achieve an improvement over standard `O`*-level flags?* Our attempt at finding improvement was unfruitful, as the handpicked flags configuration used in *ast_opt_l1* did not outperform other configurations. However, this setup was constrained by how we influence LLVM opt passes. Further work may study distinct LLVM optimization passes with a more sophisticated approach, for instance with a fork of clang and a custom LLVM Pass Manager.

(3) How much performance gain can we get from combining multiple fuzzing runs on different output binaries? As mentioned in the previous research question, we attempted to leverage the number of unique edges found to pick a combination of flags. This also did not result in any significant differences overall, but we believe this area is worth exploring more since there are configurations with higher pairwise differences than O3 and O2. Additionally, this may become even more interesting after longer runs, when a certain coverage threshold is reached and no more progress is made.

Overall, we noticed a delicate trade-off between runtime performance and code complexity emitted by different compiler flags, and hope to have laid the foundation for more work on the influence of compiler optimizations on fuzzing.

REFERENCES

[n.d.]. https://google.github.io/clusterfuzz/

[n.d.]. Add callsitesplitting Passclosedpublicactions. https://reviews.llvm.org/D39137

[n.d.]. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/. Accessed: 2022-01-06.

[n.d.]. Discussion on influencing LLVM opt flags in clang. https://lists.llvm.org/pipermail/llvm-dev/2015-December/092939.html

[n.d.]. LLVM's analysis and transform passes¶. https://llvm.org/docs/Passes.html

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. 2018. Fot: A versatile, configurable, extensible fuzzing framework. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 867–870.

Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* 7, 1 (2006), 1–30. http://jmlr.org/papers/v7/demsar06a.html

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. *AFL++: Combining Incremental Steps of Fuzzing Research.* USENIX Association, USA.

Fuzzbench. 2020. Fuzzbench report. https://www.fuzzbench.com/reports/sample/index.html

Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

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

Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*. 435–445.

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*. 2123–2138.

C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

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 *Proceedings of the 30th USENIX Security Symposium*.

Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. 2021. InstruGuard: Find and Fix Instrumentation Errors for Coverage-based Greybox Fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 568–580. https://doi.org/10.1109/ASE51524.2021.9678671

LLVM. 2022. Clang documentation. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

Barton Miller, Mengxiao Zhang, and Elisa Heymann. 2020. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering* (2020).

Barton P Miller, Gregory Cooksey, and Fredrick Moore. 2006. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st international workshop on Random testing*. 46–54.

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

Laurent Simon and Akash Verma. 2020. Improving Fuzzing through Controlled Compilation. In *2020 IEEE European Symposium on Security and Privacy (EuroS P)*. 34–52. https://doi.org/10.1109/EuroSP48549.2020.00011

András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101 arXiv:https://doi.org/10.3102/10769986025002101

7 APPENDIX

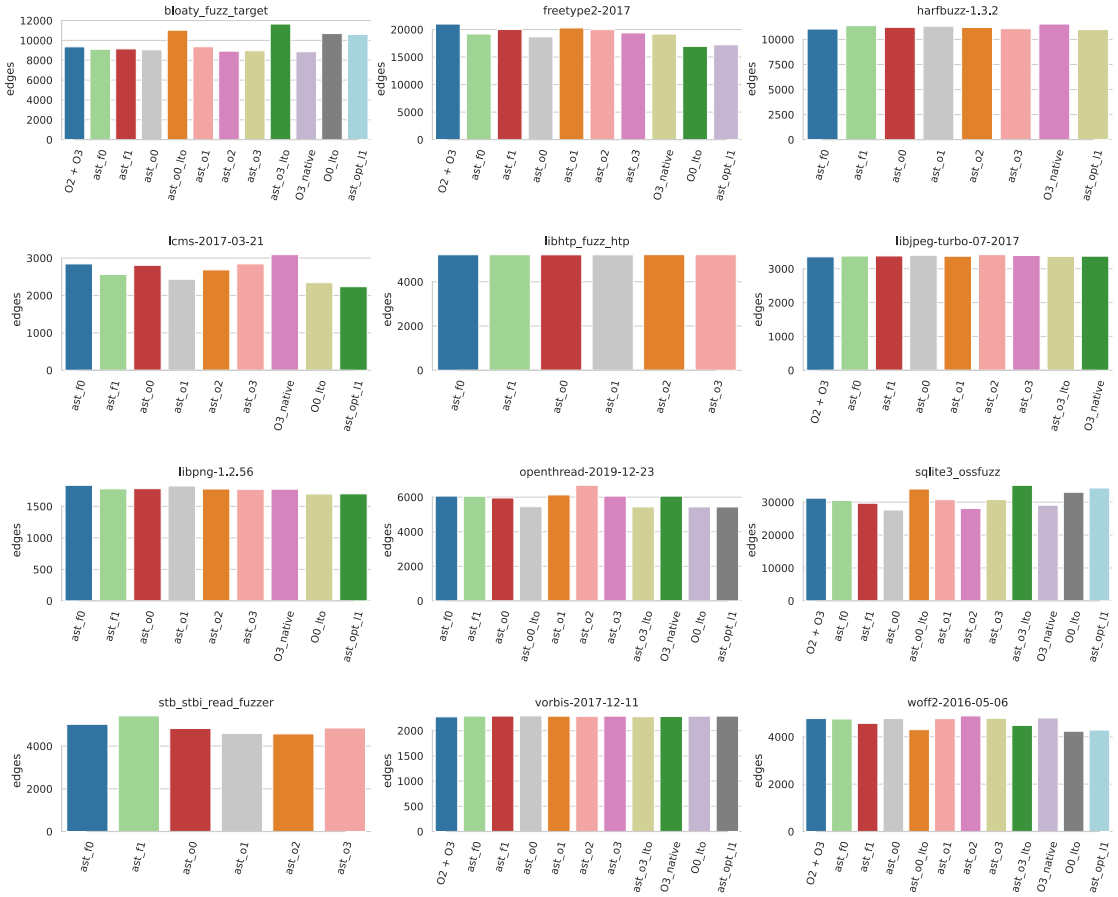

Fig. 15. Overview of edge coverage achieved after 4 hours of runtime with 3 trials each. This figure shows no fuzzer consistently outperforms any other and there are no bench-suite wide outliers (few benchmark/fuzzer pairs may be missing due to time constraints).

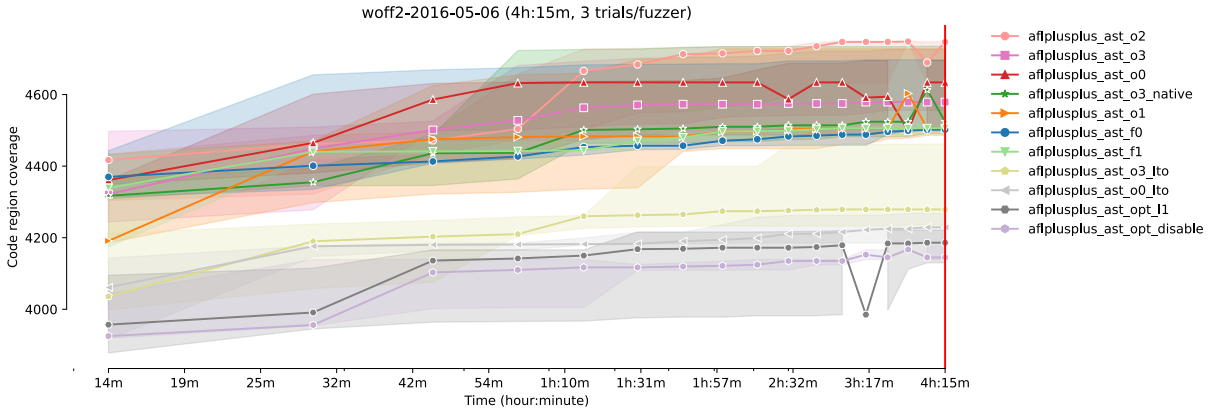

Fig. 16. Coverage growth plot for woff2-2016-05-06

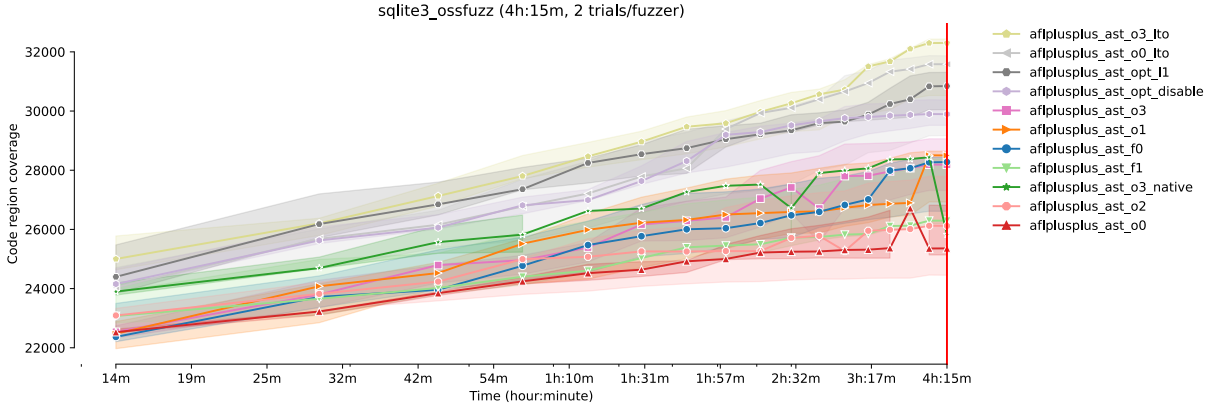

Fig. 17. Coverage growth plot for sqlite3_ossfuzz.

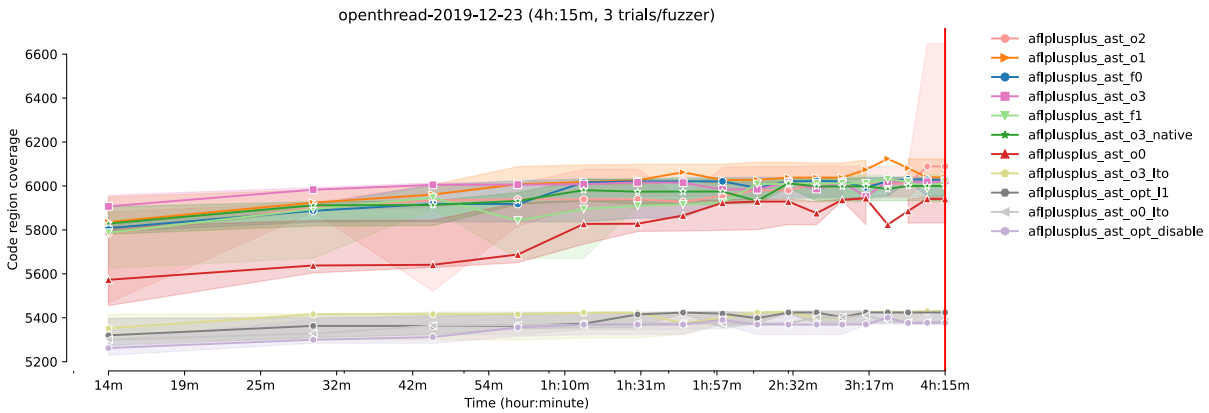

Fig. 18. Coverage growth plot for openthread-2019-12-23.

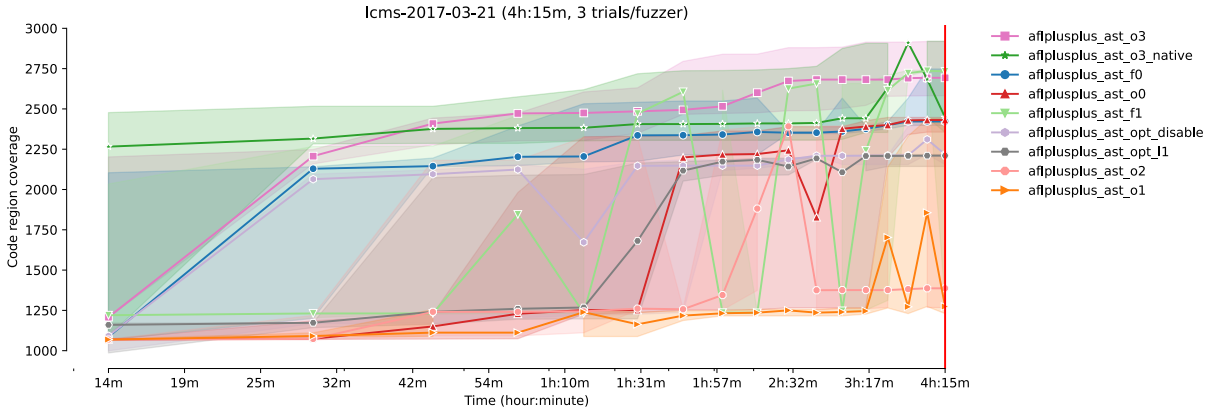

Fig. 19. Coverage growth plot for lcms-2017-03-21.

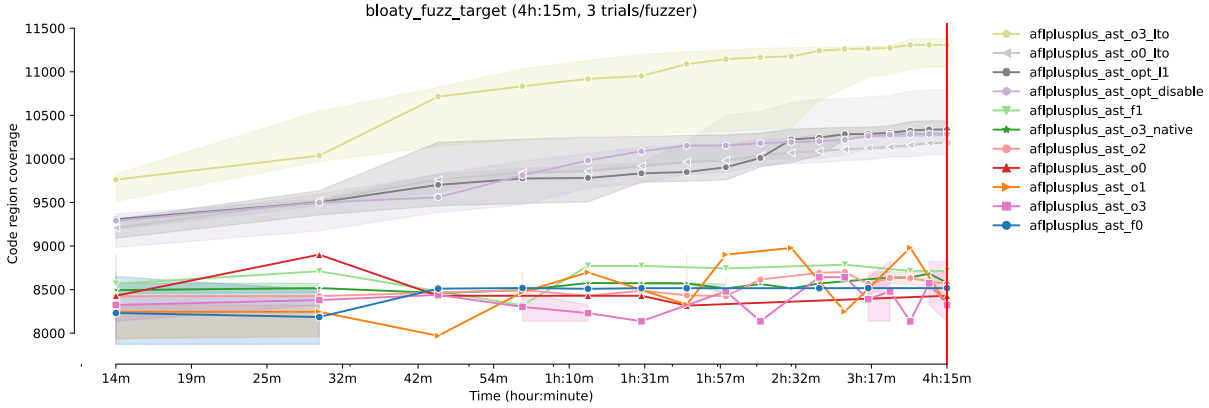

Fig. 20. Coverage growth plot for bloaty_fuzz_target.

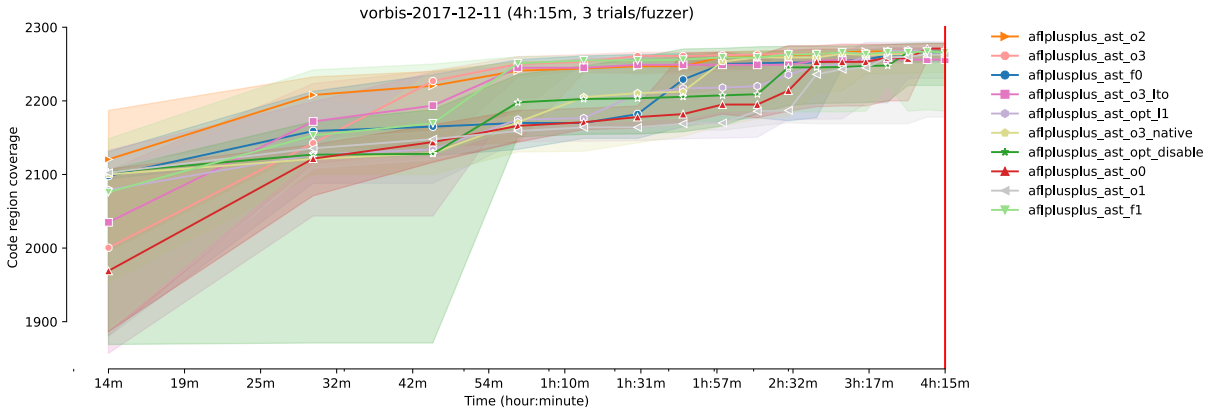

Fig. 21. Coverage growth plot for vorbis-2017-12-11.

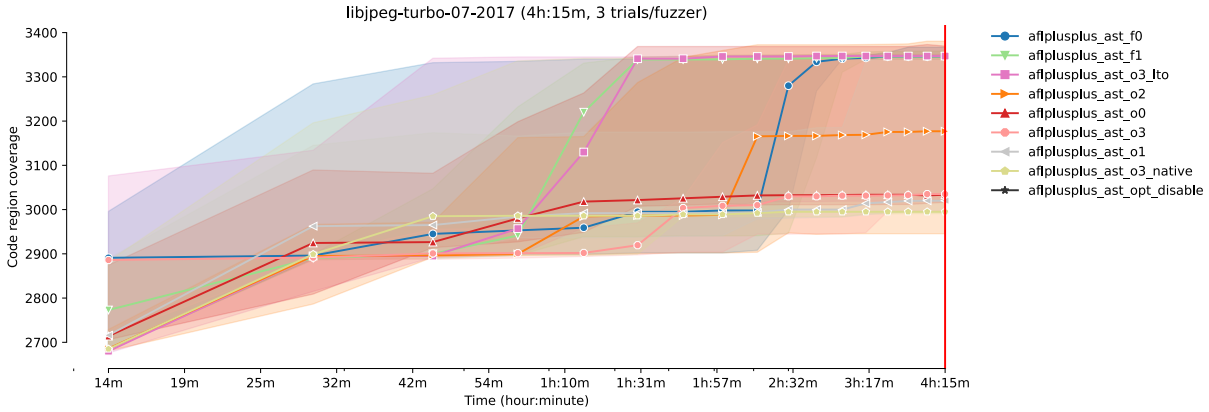

Fig. 22. Coverage growth plot for libjpeg-turbo-07-2017.

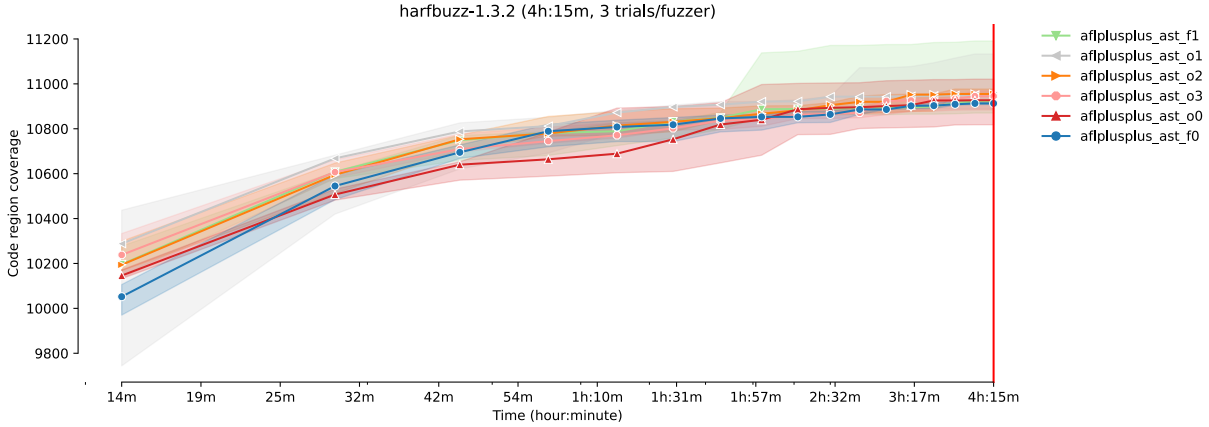

Fig. 23. Coverage growth plot for harfbuzz-1.3.2.

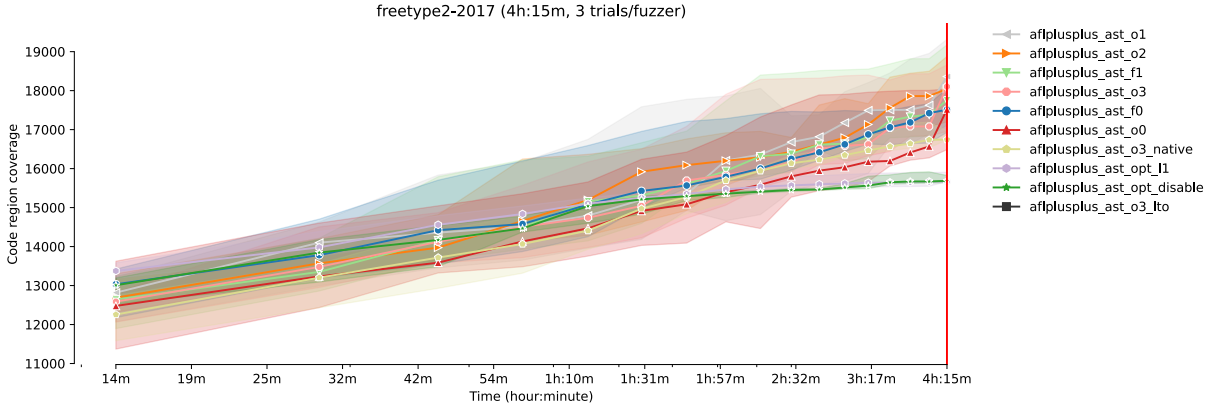

Fig. 24. Coverage growth plot for freetype2-2017.

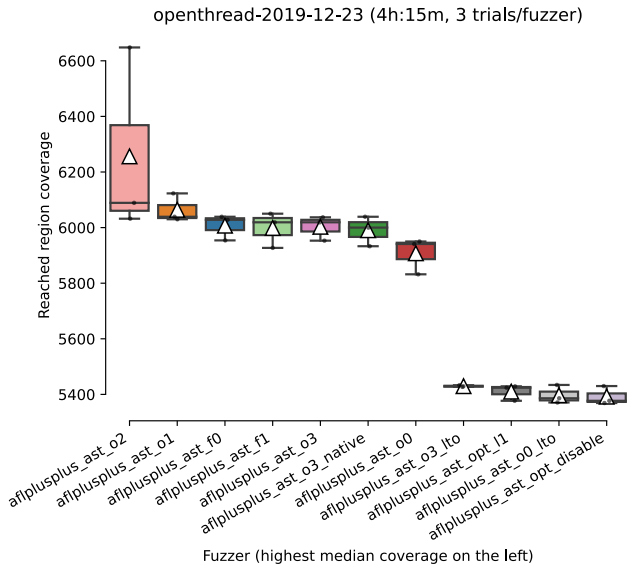

Fig. 25. openthread-2019-12-23 box plot.

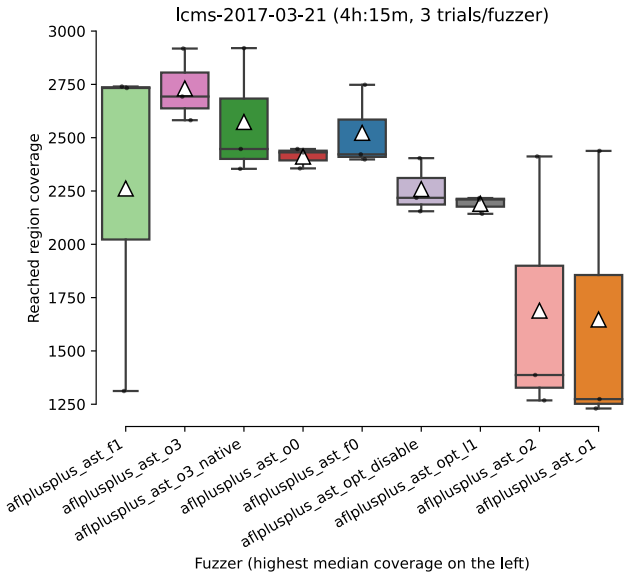

Fig. 27. lcms-2017-03-21 box plot.

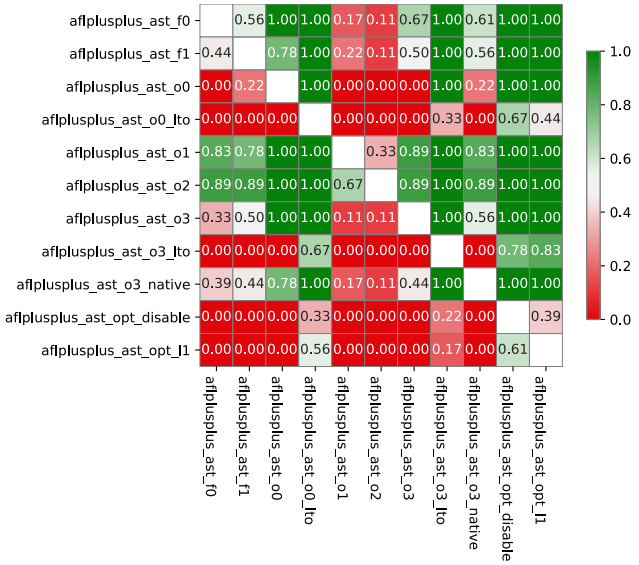

Fig. 26. Varga delaney plot for openthread-2019-12-23.

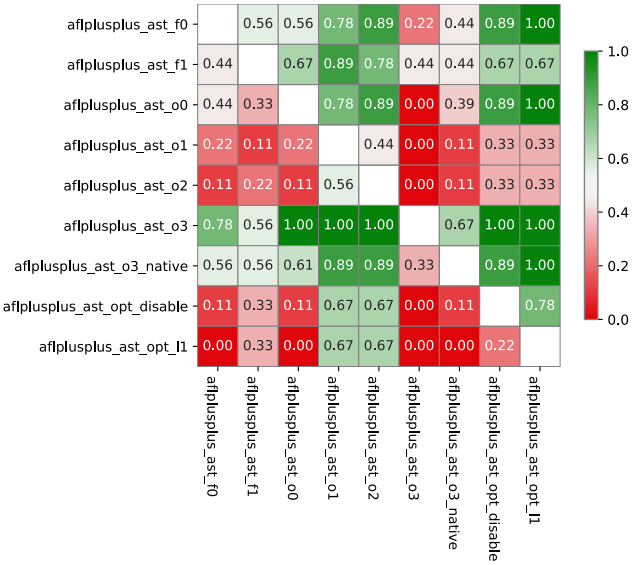

Fig. 28. Varga delaney plot for lcms-2017-03-21.

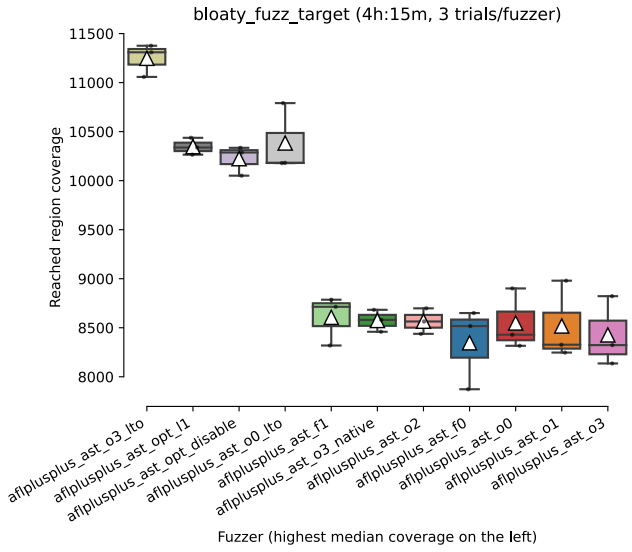

Fig. 29. bloaty_fuzz_target box plot.

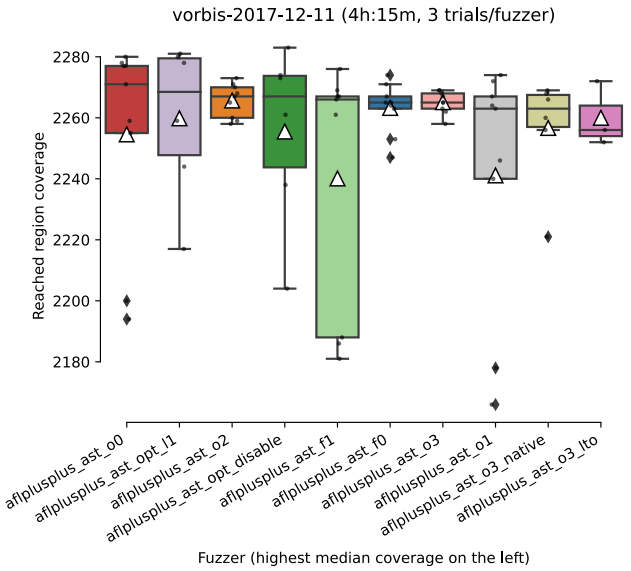

Fig. 31. vorbis-2017-12-11 box plot.

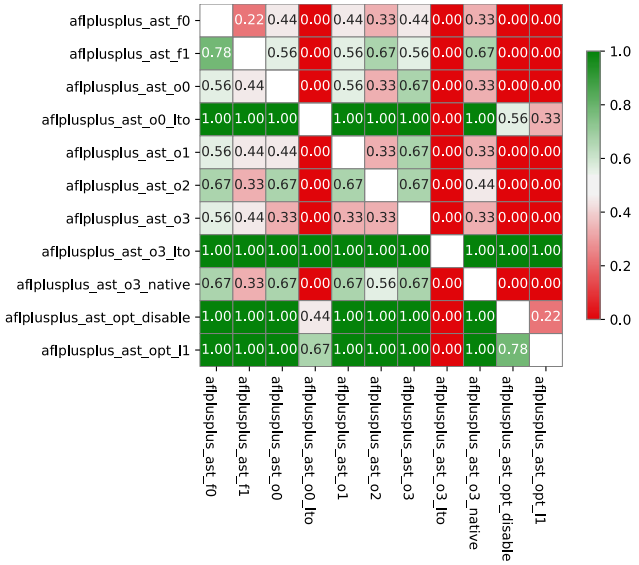

Fig. 30. Varga delaney plot for bloaty_fuzz_target.

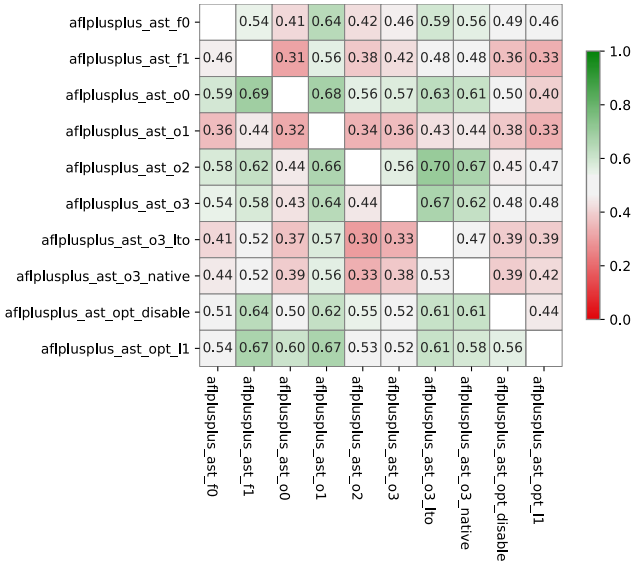

Fig. 32. Varga delaney plot for vorbis-2017-12-11.

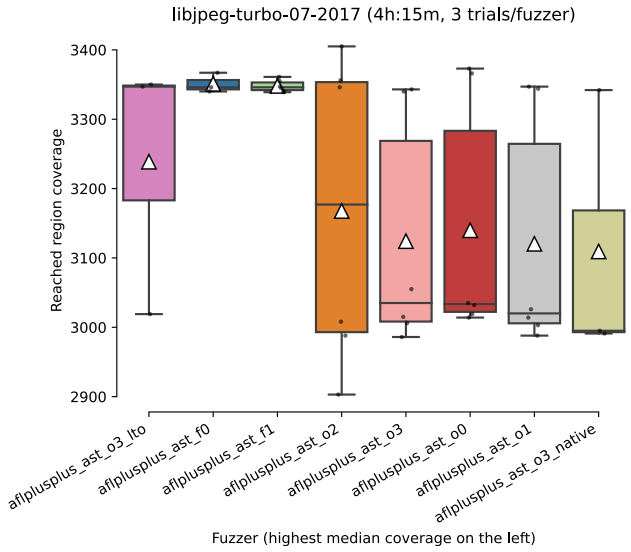

Fig. 33. libjpeg-turbo-07-2017 box plot.

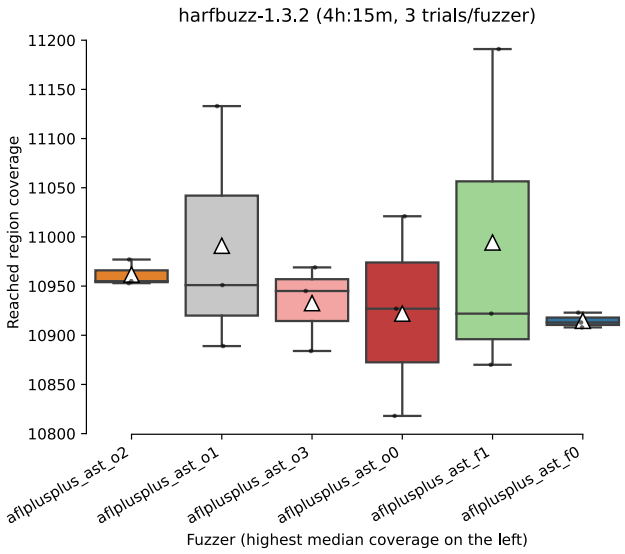

Fig. 35. harfbuzz-1.3.2 box plot.

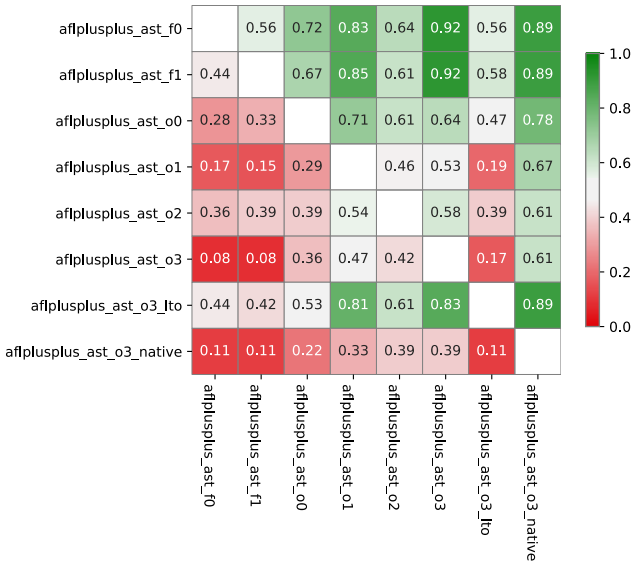

Fig. 34. Varga delaney plot for libjpeg-turbo-07-2017.

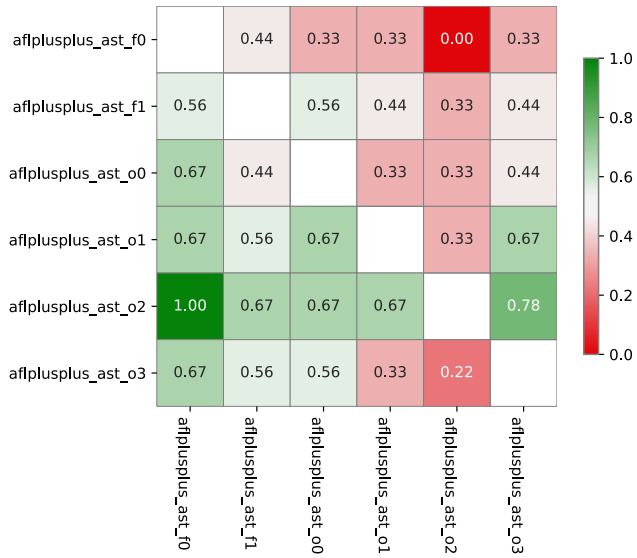

Fig. 36. Varga delaney plot for harfbuzz-1.3.2.

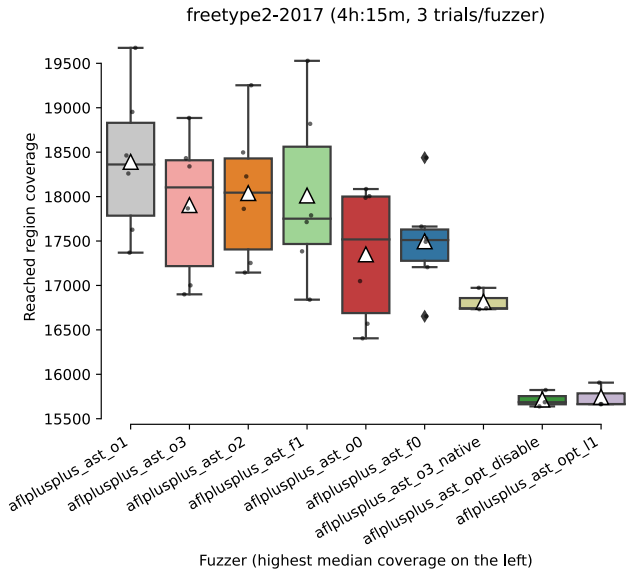

Fig. 37. freetype2-2017 box plot.

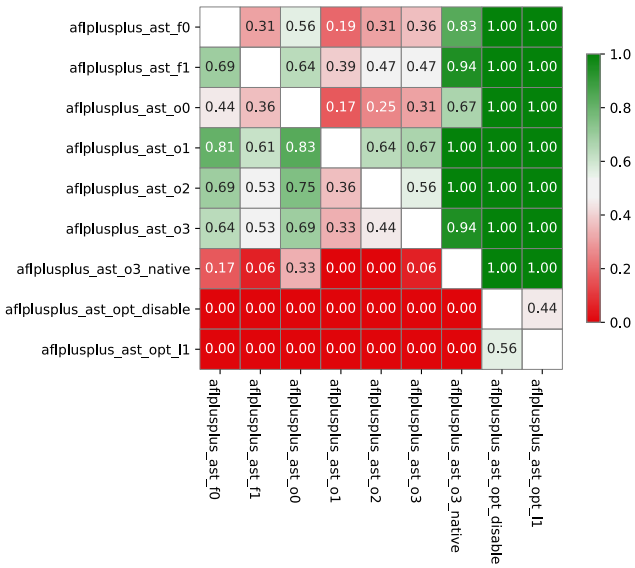

Fig. 38. Varga delaney plot for freetype2-2017.