Study and Comparison of General Purpose Fuzzers

Arijit Pramanik, Ashwin Tayade

University of Wisconsin-Madison

All binaries are available at /u/t/a/tayade/Fuzzers.

1 Abstract

Fuzz testing is a widely used technique for detection of vulnerabilities whose popularity has led to the development of various tools that do fuzz testing. General-purpose fuzzers work in all domains while some other fuzzers are targeted towards some specific domain. Evaluation of these tools is not an easy task since different fuzzing tools excel in different domains. In this paper, we evaluate 3 such general-purpose fuzzing tools namely libFuzzer, American Fuzzy Lop(AFL) and honggfuzz on 2 metrics, i.e. their bug finding capability and their code coverage. We use the google fuzzer-test-suite which has 24 applications spanning several domains. libFuzzer performs best out of the three in finding memory leaks and out-of-memory related bugs but for other kinds of bugs, all three perform at par. honggfuzz seems to be the best in terms of coverage, though libFuzzer is not far behind, which we believe is because of our runs being of short duration.

2 Introduction

Fuzz testing [3] is a powerful software testing technique to find critical bugs that might be difficult to find by other methods. It involves providing invalid, unexpected, or random data as inputs to a target application [15] which is then subsequently monitored for exceptions like overflows, failing assertions or memory leaks. There are many ways to categorize fuzzers[12] based on their strategies, their purpose and the way they do the actual fuzzing and process monitoring.

Based on their purpose, fuzzers can be general purpose or they can be attuned to perform well in certain domains. General purpose fuzzers should perform decently well in finding bugs in all kinds of applications whereas specialised fuzzers must be very good at finding bugs in applications of specific domain at the cost

of not performing well in applications that lie outside this domain. We study only general purpose fuzzers here.

While each fuzzer has its own strategy to generate inputs, there are two general strategies. One is the generation based strategy, that uses a context-free grammar or an input model as a specification to generate input. Integrating fuzzing with such input and state models can make fuzzing more effective. The other strategy is mutation based, in which fuzzers modify existing test cases to generate new inputs. In this paper, we compare mutation based fuzzers, though we believe that some of the bugs in the applications might be easier to find using generation based fuzzers.

Code coverage[1] is a metric used to judge the degree to which the source code of a program is executed for an application. This is useful because if more of the source code is executed during testing, then it has a lower chance of containing software bugs compared to a program with low coverage. Motivated by this, fuzzers can also be categorized as being black-box, grey-box or white-box depending on what kind of knowledge of the program structure they require to perform. A black-box fuzzer is completely unaware of the internal program structure, while a white-box fuzzer leverages program analysis to systematically increase code coverage or to reach certain critical program locations. A grey-box fuzzer leverages instrumentation rather than program analysis to gain information about the program. All three fuzzers compared here are grey-box fuzzers that use instrumentation while compiling the binary from the source code. We also discuss about a black-box fuzzer which stands in contrast to other grey-box fuzzers studied above.

We select 3 fuzzers which are general purpose, mutation based grey-box fuzzers namely libFuzzer [18], American Fuzzy Lop[25] and honggfuzz[21] each of which are explained in further sections. We also study and discuss the impact of using a mutation-based black-box fuzzer, Radamsa [11].

3 Related Work

Two widely explored approaches in fuzz testing are generation based fuzzing and mutation based fuzzing. Mutation based fuzzers generate test cases by modifying the seed, which is typically a well-structured input.

3.1 Advances in fuzzing in the industry

In the industry, fuzz testing software has taken rapid strides. For example, Google's OSS-Fuzz[19] platform has found more than 1000 bugs in 5 months with thousands of virtual machines[5]. Different methods have been proposed in different papers which aim to strengthen different aspects of fuzzing.

- **Improve Seeds**: SkyFire[23] and Orthrus[20] improve seeds by performing an initial analysis to gain information about the seeds. QuickFuzz[9] generates seeds using a predefined context free grammar.
- **Mutation of inputs**: This is one place where a lot of methods have been proposed. SYMFUZZ [6] uses a symbolic executor to determine the number of bits to mutate. We discuss this in the following subsection. A lot of fuzzers have also started integrating machine learning techniques to select mutations, mutation ratios and mutation sites. FuzzerGym [7] uses reinforcement learning with libfuzzer to select mutation operations.
- **Classifying an input as interesting**: Most papers focus on crashes as their primary criterion, but few studies have used different criteria to change what interesting means; like longer running time, using more resources etc. SlowFuzz[16] prioritizes seeds that use more computer resources (e.g., CPU, memory and energy). Dowser[10] and VUzzer[17] use static analysis to assign rewards to points in a program for reaching a deeper point in the control flow graph. This loosely means that coverage is used as a metric.
- **Choosing the next input to execute**: This is another place where algorithms majorly differ from one another. Some maintain a queue, while some randomly pick the next input. Others use scores to choose what they believe is the best input. For example, AFL's approach is based on branch coverage with a logarithmic counter on each branch, which allows branch counts to be considered different only when they differ in orders of magnitude. honggfuzz computes coverage based on the number of executed instructions, executed branches, and unique basic blocks. This metric allows the fuzzer to add longer executions, which can help discover denial of service vulnerabilities or performance problems.

3.2 Comparison of fuzzers

The evaluation of free fuzzing tools [22] uses a set of diverse fuzzers Radamsa, Microsoft MiniFuzz, JBroFuzz, Burp suite, w3af and ZAP to compare, but not all of these are popular. A review of fuzzing tools and methods [8] reviews many fuzzing tools with a focus on how the tools work. Evaluating Fuzz Testing[14] cites many papers where fuzzer comparison is done, like Designing New Operating Primitives to Improve Fuzzing Performance[24] and QuickFuzz[9], using unique bugs as a metric but miss using the ground truths, which list all known bugs that exist in the applications. We overcame this problem by using the google fuzzer-test-suite [4] which tells us what the fuzzers need to look for.

4 Fuzz Tools

In this section we explain how each of the fuzzers work and the strategies used by them. The general algorithm of a mutation-based fuzzer [13] is as follows. Each fuzzer differs in its implementation of `select_mutation`, `select_mutation_site` and `mutate` functions.

Algorithm 1: Fuzzing algorithm in mutation based fuzzers

```
time: Fixed time window to fuzz
queue: Queue of inputs that may find new paths
while time!=0 do
    parent, energy ← get_input_from_queue()
    while i ≤ energy do
        child ← parent
        j ← 1
        while j ≤ num_mutations do
            mutation ← select_mutation()
            site ← select_mutation_site()
            child ← mutate(mutation, site)
        end
        path ← run_code(child)
        if path is new then
            queue.add(child)
        else
            continue
        end
    end
    decrease time
end
```

4.1 American Fuzzy Lop(AFL)

American Fuzzy Lop [25] is a sophisticated but easy to use fuzzer that uses instrumentation and genetic algorithms to find test cases that execute new code paths. The target binary is first compiled with instrumentation. Then, the provided seed input is used to run the binary. Interesting inputs (primarily which increase

code coverage) are identified and further mutations are done on these inputs preferentially. The possible mutations done by AFL are shown in Table 1. By this preferential treatment, the test cases that execute new code paths get more execution time. Crashes and hangs which are detected by SIGABRT, SIGILL, SIGBUS, SIGSEGV and SIGABRT signals, are written to the output directory. AFL is very sensitive to input seed corpus. Large seed test cases and large number of seeds negatively affect its performance. Even ordering of the input seeds impacts its performance. AFL also has a lot of different modes which can be used for analysis and test corpus minimisation. All these modes are experimental and sometimes do not perform as expected. Some of which are:

- afl-tmin: It is a test case minimizer that goes through the input file and identifies the parts of the input that explore new execution paths. It then creates a much smaller version of the input file.
- afl-cmin: This mode goes through the entire corpus and discards the test cases that do not trigger any new states/paths. It does not truncate any test case, but minimises the number of test cases in the corpus.
- afl-analyze: This is a syntax analyzer mode that classifies the bytes in the test case as being no-op, critical etc.

In addition, AFL can also be run in parallel mode where each instance periodically scans the synced test case directory for any test cases found by other fuzzer instances.

4.2 libFuzzer

libFuzzer[18] is a coverage guided, mutation based, grey-box fuzzer. It tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage. Coverage is counted as the number of basic blocks it hits. libFuzzer can be used with/without seed inputs. This is a major advantage in cases where we might lack the availability of seed inputs. Execution when begins with a seed input, a mutator is selected and applied, and a new input is created and tested. This new input might cause either a rewarding outcome(more coverage) or not (no new coverage). If the new input causes an increase in coverage, it is added to the input corpus to be selected in the future, otherwise it is discarded. Each new input is generated using a combination of operations [7] shown in Table 2. Crashes are written in the form of reproducers. libFuzzer in parallel uses multiple threads to exchange the corpora. Given that this corpora can become large pretty quickly, libFuzzer has a corpus minimisation mode that can

Operation	Effect
Bitflips	Flip single bit
Interesting Values	NULL, -1, 0, etc.
Addition byte	Add random value
Subtraction	Subtract random value
Random Value	Insert random value
Deletion	Delete from parent
Cloning	Clone/add from parent
Overwrite	Replace with random
Extra Overwrite	Extras: strings scraped from binary
Extra Insertion	Extras: strings scraped from binary

Table 1: AFL Mutation operations

reduce the test inputs in the corpus while preserving coverage. It also uses LeakSanitizer present in llvm to detect memory leaks and out-of-memory bugs, with the help of a leak detection phase at the end of each run.

4.3 honggfuzz

honggufuzz [21] is a coverage-guided mutation based grey-box fuzzing tool. It uses the POSIX signal interface to detect and log crashes, which introduces the drawback of using a POSIX-compliant operating system. It can use both compile-time and sanitizer-coverage instrumentation on the target binary. The biggest advantage of honggfuzz is that it can utilize hardware-based counters and Branch Trace Store and Processor Tracing from Intel for coverage if the CPU supports them. Given an initial input corpus, it identifies files which add new code coverage or increased instruction/branch counters and adds them to an in-memory dynamically stored corpus. Then it randomly chooses files from this corpus, mutates them and begins a new fuzzing round. If this newly created file induces new code paths, then it is added to the corpus as well. It uses the *ptrace* API in Linux to detect SIGABRT, SIGILL, SIGBUS, SIGSEGV and SIGABRT signals using associated signal handlers.

Operation	Effect
EraseBytes	Reduce size by removing a random byte
InsertByte	Increase size by one random byte
InsertRepeatedBytes	Increase size by adding at least 3 random bytes
ChangeBit	Flip a Random bit
ChangeByte	Replace byte with random one
ShuffleBytes	Randomly rearrange input bytes
ChangeASCIIInteger	Find ASCII integer in data, perform random math ops and overwrite into input.
ChangeBinaryInteger	Find Binary integer in data, perform random math ops and overwrite into input.
CopyPart	Return part of the input
CrossOver	Recombine with random part of corpus/self
AddWordPersistAutoDict	Replace part of input with one that previously increased coverage
AddWordTempAutoDict	Replace part of the input with one that recently increased coverage
AddWordFromTORC	Replace part of input with a recently performed comparison

Table 2: libFuzzer Mutation operations

4.4 Radamsa

Radamsa [11] is a general-purpose, mutation based fuzzer. It works by reading valid sample input files and generating different outputs from them by mutation while trying to keep the general input format somewhat valid. Radamsa is an extreme black-box fuzzer; it needs no information about the internals or source code of the program nor the input format of the data. Radamsa was developed as a tool to test how well a program can withstand malformed and potentially malicious inputs. On the bright side, it is easy to set up and run and has already been successful in finding bugs in several programs. It can be paired with coverage analysis tools during testing to improve the quality of the sample input set during a continuous fuzzing test. This helps in comparison with other fuzzers on the basis of edge-coverage, branch-coverage and other metrics. The black box approach limits the rate with which Radamsa can penetrate deep into the tested system, especially compared to other fuzzers employing instrumentation.

5 Google Fuzzer Test Suite

Overall, fuzzing performance may vary with the target program, so it is important to evaluate on a diverse, representative benchmark suite. We have selected the google fuzzer-test-suite which has applications spanning various domains like image processing, SQL, json parsing, etc. This test suite is inspired by real-life applications that have known bugs, hard-to-find code paths, or other challenges for bug finding tools. Each of the 24 applications in the suite has some version of the application which has memory-leaks, crashes or input-agnostic errors like assertion failures and lines of code difficult to reach. Some of them have a set of specific inputs, or seeds to help the tools in finding bugs and maximising coverage. We now explain the applications in the test suite. We also mention the CVE[2] (a list of publicly known cybersecurity vulnerabilities) identification number for some notable bugs like multi-byte-read-heap-buffer-overflow(HeartBleed (CVE-2014-0160)) in openssl-1.0.1f, 1-byte-write-heap-buffer-overflow (CVE-2016-5180) in c-ares-1.x and a heap-buffer-overflow (CVE-2018-5146) in vorbis on Firefox.

- **boringssl**: It is a software library that secures connection over computer networks. It is the OpenSSL equivalent developed for specific use by Google.
- **c-ares**: It is a C library for asynchronous DNS requests.
- **freetype2**: It is a font engine used to efficiently produce glyphs.
- **guetzli**: It is a freely licensed JPEG encoder developed at Google. We were unable to compile the binary for this application.
- **harfbuzz**: It is an engine used to convert unicode to glyphs.
- **json**: Library to process JSON.
- **little-cms**: It is an open source color management system.
- **libarchive**: It is an open-source C programming library that provides access to a variety of different archive formats like tar, zip etc.
- **libjpeg-turbo**: It is an image codec that is used for faster JPEG compression and decompression.
- **libpng**: It is a library of C functions to handle PNG images.
- **libssh**: It is C library implementing the SSHv2 protocol on client and server side.

- **libxml2**: It is the XML C parser and toolkit.
- **llvm**: A C++ demangler.
- **openssl**: OpenSSL is a Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocol.
- **openthread**: It implements all thread networking layers.
- **pcre2**: It is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.
- **proj4**: It is a coordinate transformation library.
- **re2**: It is a software library for handling regular expressions.
- **sqlite**: It is a relational database management system contained in a C library.
- **vorbis**: It is a software library used to produce lossy audio encoding.
- **woff2**: It is a web font compression library.
- **wpantund**: It is a network interface daemon that provides a native IPv6 network interface to a low-power wireless Network Co-Processor.

6 Implementation

The preferences and fuzzing strategies of fuzzers vary greatly across several applications, though in all cases, all fuzzers are run at full capacity. We run the fuzzers in parallel mode which effectively improves the performance of the fuzzer. Here, we are running multiple base fuzzers on all cores. However, these suffer from a lack of diversity when the same fuzzing strategy is used across all instances although each fuzzer instance has access to the corpora of inputs from other instances. For certain applications, we can build an input dictionary (a set of predefined values with potentially significant semantic weight) by statically analyzing program control and data flow. This influences the input generation and helps increase the fuzzer's effectiveness. We did 5 runs, trying to make each one last three hours on each application that we could build.

- AFL: AFL runs in a master slave model when we run parallel threads of AFL on a binary. A master thread is first started, followed by any number of slave threads. All the threads write outputs which

execute new code paths to the same directory thus helping each other. The difference between a master and a slave thread is that the master makes deterministic choices while a slave might run randomly. This makes the slaves behave in a happy go lucky manner while the master behaves in a systematic ways. We ran AFL with one master and 7 slaves. Parallel running of AFL should in theory make for a better fuzzing technique, but we believe that this is true only when the fuzzing is done in long extended runs. In shorter runs, parallel threads increase the queue of inputs originating from one seed making it very large. This leads to the other seeds not getting enough attention.

- libFuzzer: libFuzzzer considers all running threads the same. All threads run in a systematic manner making deterministic choices. All the threads write to the same corpus directory, thus helping the other threads. We ran 8 threads of libFuzzer for testing. Each of the threads stop after a crash is detected. If the bug is shallow, all threads stop fairly quickly. In such cases, we could not run the fuzzer for the expected 3 hours.

- honggfuzz : honggfuzz is multi-threaded and hence we ran 8 threads of honggfuzz. The input corpus is shared among all the threads automatically and is improved by adding test cases that increase coverage, wherein each thread can benefit from one another. The parallel fuzzing model is similar to libFuzzer. It can also be run in a persistent fuzzing mode where we run each new input within the same process which reduces the overhead of starting a new fuzzing process. Since the source code is available for compile-time instrumentation, we did not take advantage of hardware counters to probe increase in coverage.

- Radamsa : Radamsa comes as a simple binary which can be run on multiple threads, each generating its own set of inputs. We set up Radamsa in a loop, fed its output to the system under test to detect bugs. We ran 8 parallel instances of the same. We found that there is no shared input corpus and fuzzing is not effective as each binary might generate and use the same set of inputs.

7 Evaluation

We used an Intel(R) Core(TM) i7-8550U CPU machine with 4 physical cores running Ubuntu 16.04.5 LTS for all our experiments. In this section we present the results of our experiments in tabular form for each of the applications in the google fuzzer-test-suite across all 3 fuzzers viz, libFuzzer, AFL and honggfuzz. We show the bugs found and the code coverage in terms of basic blocks, edges and program counter. We can't

directly compare Radamsa to the above 3 fuzzers since it does not use any execution feedback like basic block coverage to mutate test cases with the aim of higher probability of reaching unexplored code blocks.

7.1 Unique bugs found

We show the number of bugs that the particular fuzzer caught in a particular application split between Table 3, Table 4 and Table 5. We also show how many bugs are present in the application, acting as the ground truth. We see that libFuzzer caught most of the bugs as the other two fuzzers. One particularly interesting find was that AFL and honggfuzz missed the bugs that were caused by memory leaks because such bugs do not result in signals like SIGSEGV, SIGABRT, etc which can be detected by AFL and honggfuzz. However, libFuzzer caught them owing to LeakSanitizer. Similarly, out-of-memory bugs were caught by libFuzzer but they were missed by AFL and honggfuzz. It seems that in small multiple runs, libFuzzer performs somewhat better than AFL and honggfuzz. We observed that AFL is affected by the order of the seeds and modifying the order of seed inputs affected its performance. This is not a very good implementation in case the first seed is not very good. In such a case, AFL will spend a lot of time executing already seen paths. Hence, quality of seeds is very important for AFL. libFuzzer and honggfuzz are unaffected by the order of seeds but their ability to explore new unexplored code paths still affects them. We also observed that performing fuzzing with dictionaries helped find vulnerabilities in a much lesser time. Now we discuss some interesting findings.

- An interesting case was observed in pcre2, where we found more bugs than actually reported by the test suite. More testing on the latest version is needed to investigate the status of this unreported stack-overflow bug.
- libxml2 had a shallow bug, which caused a deeper bug to be missed. We believe that longer runs can take care of this problem. honggfuzz could not find any bug in this application.
- re2 has a "DFA out of memory" issue which is reported in the logs of libFuzzer without crashing. AFL and honggfuzz missed this bug since a crashing signal was not sent by the application.
- woff2 hits the out-of-memory bug first when run with an empty corpus. However, when initialized with the given corpus of seed inputs, it finds the heap-buffer-overflow bug first, which demonstrates the importance of input seeds to start fuzzing.

- For harfbuzz and openssl1.0.1c, libFuzzer could not find any of the bugs, which were found by both AFL and honggfuzz.

Application	**AFL**	**libFuzzer**	**honggfuzz**	**Bugs present**
boringssl	1 (heap-use-after-free)	1 (heap-use-after-free)	1 (heap-use-after-free)	1 (heap-use-after-free)
c-ares	1 (heap-buffer-overflow)	1 (heap-buffer-overflow)	1 (heap-buffer-overflow)	1 (heap-buffer-overflow)
freetype2	0	0	0	0
guetzli	N/A	N/A	N/A	1 (assertion failure)
harfbuzz	1 (assertion failure)	0	1 (assertion-failure)	1 (assertion failure)
json	1 (assertion failure)	1 (assertion failure)	1 (assertion failure)	1 (assertion failure)
little-cms	0	1 (heap-buffer-overflow)	0	2 (2 heap-buffer-overflows)
libarchive	N/A	0	0	1 (heap-buffer-overflow)
libjpeg-turbo	0	0	0	0
libpng	0	1 (out-of-memory)	0	1 (out-of-memory)
libssh	0	1 (memory leak)	0	1 (memory leak)
libxml2	2 (2 heap-buffer-overflows)	2 (2 heap-buffer-overflows)	0	4 (3 heap-buffer-overflows, 1 heap-use-after-free)

Table 3: Unique bugs found for each fuzzer in each application

7.2 Code coverage

Table 6 shows the basic blocks covered by AFL along with edges covered in the Control Flow Graph(CFG) of the program for libFuzzer and honggfuzz in each application. We believe this is a fair comparison since more edges covered implies more basic blocks visited and vice versa. Though strictly they aren't the same, they are highly correlated. honggfuzz provides other insights such as number of branch statements executed; while libFuzzer provides cumulative coverage data from edge counters, value profiles, indirect caller/callee pairs, etc. No single fuzzer individually provides all different coverage metrics, hence we have presented our

Application	AFL	libFuzzer	honggfuzz	Bugs present
llvm-libcxxabi	1 (stack-overflow)	1 (stack-overflow)	1 (stack-overflow)	2 (1 stack-overflow, 1 out-of-range access)
openssl1.0.1f	1 (heap-buffer-overflow)	1 (heap-buffer-overflow)	0	2 (1 heap-buffer-overflow, 1 memory leak)
openssl1.0.2d	1 (assertion failure)	1 (assertion failure)	1 (assertion failure)	1 (assertion failure)
openssl1.1.0c	1 (heap-buffer-overflow)	0	1 (heap-buffer-overflow)	3 (1 heap-buffer-overflow, 2 non-reproducible)
openthread	0	0	0	12 (10 stack-buffer-overflows, 1 heap-buffer-overflow, 1 null dereference)
pcre2	2 (1 heap-buffer-overflow, 1 heap-use-after-free)	3 (1 heap-buffer-overflow, 1 heap-use-after-free, 1 stack-overflow)	1 (1 heap-buffer-overflow)	2 (1 heap-buffer-overflow, 1 heap-use-after-free)

Table 4: Unique bugs found for each fuzzer in each application

comparison only on the basis of basic code blocks/edges covered in a Control Flow graph. Since, AFL is affected by the seeds, the coverage is inferior to libFuzzer in case of smaller runs. AFL spends a lot of time on initial seeds and the run ends with an insufficient number of seeds explored. libFuzzer outperforms AFL in this metric. We can clearly see that the code coverage achieved by the runs is low. honggfuzz performs better than both for most of the cases. This is an indication that the test suite is geared towards finding bugs as a metric. The seeds provided hit those code paths that have bugs, but do less in terms of increasing coverage. We discuss some interesting cases below.

- woff2 coverage of libFuzzer is low for an empty seed corpus because of a shallow out-of-memory bug

Application	AFL	libFuzzer	honggfuzz	Bugs present
proj4	0	2 (2 memory leaks)	0	2 (2 memory leaks)
re2	1 (heap-buffer-overflow)	2 (1 out-of-memory, 1 heap-buffer-overflow)	1 (heap-buffer-overflow)	2 (1 out-of-memory, 1 heap-buffer-overflow)
sqlite	0	0	0	3 (2 heap-buffer-overflows, 1 memory leak)
vorbis	0	1 (heap-buffer-overflow)	3 (2 heap-buffer-overflows, 1 null dereference)	3 (2 heap-buffer-overflows, 1 null dereference)
woff2	1 (heap-buffer-overflow)	2 (1 heap-buffer-overflow, 1 out-of-memory)	1 (heap-buffer-overflow)	2 (1 heap-buffer-overflow, 1 out-of-memory)
wpantund	0	0	0	0

Table 5: Unique bugs found for each fuzzer in each application

which ends all threads of libFuzzer fairly quickly. But for a different set of input seeds, it hits another bug (heap-buffer-overflow) which leads to much higher coverage values. We report the latter one here.

- proj4 has a lot more coverage using libFuzzer compared to others. We could not identify the cause of this big difference.

- AFL attains comparatively higher coverage values for pcre2 and llvm-libcxxabi.

- Compared to the number of edges in the Control Flow Graph, the coverage across all fuzzers for openssl1.0.1f and openssl1.1.0c is quite low. We suspect that the input seeds guide the fuzzers towards finding bugs to a point which doesn't lead to significant increase in coverage. We also notice that honggfuzz which fails to find any bug ends up with significantly higher coverage for openssl1.0.1f.

Application	AFL	libFuzzer	honggfuzz	No. of edges in CFG
boringssl	581	1,118	1,222	14,623
c-ares	26	27	25	44
freetype2	2,649	4,877	4,468	15,898
guetzli	N/A	N/A	N/A	N/A
harfbuzz	3,792	3,975	4,878	9,292
json	297	542	1,008	1,437
little-cms	843	1,137	890	5,805
libarchive	N/A	1,967	2,481	9,864
libjpeg-turbo	1,306	1,337	1,315	8,087
libpng	552	468	632	2,878
libssh	384	703	805	6,695
libxml2	3,487	3,413	3,803	41,373
llvm-libcxxabi	2,353	1,098	1,861	2,820
openssl1.0.1f	471	479	2,370	30,933
openssl1.0.2d	603	716	779	3,663
openssl1.1.0c	899	808	759	28,945
openthread	728	1701	2,263	12,968
pcre2	12,069	2,974	3,588	8,059
proj4	489	2229	666	5,571
sqlite	1,460	5,960	9,229	17,747
re2	1,597	2,163	2,891	5,844
vorbis	763	842	988	4,613
woff2	672	953	1,861	9,836
wpantund	137	3,682	6,676	20,427

Table 6: Average number of basic blocks/edges covered by each fuzzer in each application

8 Future Work

We see three types of improvements that could be done to improve the study and comparison among fuzzers.

First is the improvement of this comparison study. We believe that it needs to be repeated with longer duration and even more number of runs. This might increase coverage and decrease the importance of seeds seen in shorter runs.

Secondly, there is a need for more generalized metrics as well as test suites to compare fuzzers across domains and fuzzing strategies. Such metrics have the potential to become the basis for evaluating new fuzzers. Better test suites are needed which take multiple metrics into account instead of concentrating on bug finding.

Thirdly, we feel that more exploration in terms of coverage should be done by making the fuzzers co-operate. For example, a corpus prepared by libFuzzer could be improved by afl-cmin and so on. The best approach for fuzzing would be to combine different base fuzzers to work in tandem and cooperate to find bugs.

9 Conclusion

Mutation based fuzzing often requires millions of executions to find bugs in real life applications. We have also seen that the these fuzzers are heavily dependent on the quality of seed inputs. Also consider for example a highly structured format, one based on SQL(the bugs in which none of the fuzzers could find). Blindly mutating a well-formed input is likely to break the structure of the input, causing it to be rejected early in processing by the program under test. This suggests that generation based fuzzers might perform better in cases with highly specific structured inputs, since they can generate valid test cases.

We found that this particular test suite is not geared towards coverage as a metric. This is because the seeds provided seem to direct the fuzzers towards code paths where the bugs exist.

We also see that honggfuzz performed the best in short multiple runs while considering coverage, with libFuzzer mostly at par with it. In the case of discovering memory leaks and out-of-memory bugs, libFuzzer is better. For other kinds of bugs, all fuzzers perform similarly. AFL provides a lot of tools for the analysis of mutated input corpus. libFuzzer suffers from the problem of terminating after finding a preset maximum number of crashes. Both of them produce crash-logs that could be used to reproduce the bug. However, honggfuzz does not provide for any such log that could be used to reproduce the crash and hence has the drawback of not being able to provide us with a set of reproducible crash files for future reference. Hence, no fuzzer is perfect. We think that fuzzers used in cooperation with one another would perform better. For example, the image below shows how each unique mode of a fuzzer can be used for superior performance.

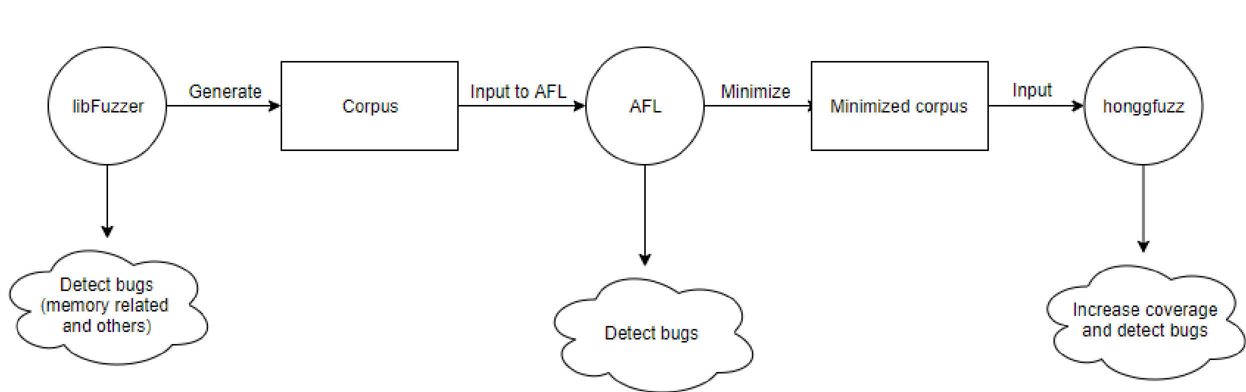

Figure 1: Example of cooperation between fuzzers

References

[1] Code coverage. URL: https://en.wikipedia.org/wiki/Code_coverage.

[2] Cve(common security vulnerabilities). URL: https://cve.mitre.org/.

[3] Fuzz testing. URL: https://en.wikipedia.org/wiki/Fuzzing.

[4] Google fuzzer-test-suite. URL: https://github.com/google/fuzzer-test-suite.

[5] Google security blog regarding oss-fuzz updates. URL: https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[6] Symfuzz. URL: http://security.ece.cmu.edu/symfuzz/.

[7] William Drozd and Michael D Wagner. Fuzzergym: A competitive framework for fuzzing and learning. *arXiv preprint arXiv:1807.07490*, 2018.

[8] James Fell. A review of fuzzing tools and methods. Technical report, Technical Report. https://dl. packetstormsecurity. net/papers/general/a . . . , 2017.

[9] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. Quickfuzz: An automatic random fuzzer for common file formats. In *ACM SIGPLAN Notices*, volume 51, pages 13–20. ACM, 2016.

[10] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the 22nd USENIX Security Symposium*, pages 49–64, 2013.

[11] Aki Helin. Radamsa, 2016. URL: https://gitlab.com/akihe/radamsa.

[12] Mohammed A Imran. Fuzzing tools. URL: https://github.com/secfigo/Awesome-Fuzzing#tools.

[13] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47. ACM, 2018.

[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[15] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[16] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168. ACM, 2017.

[17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[18] K Serebryany. libfuzzer a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[19] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. 2017.

[20] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. Static program analysis as a fuzzing aid. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 26–47. Springer, 2017.

[21] Robert Swiecki. Honggfuzz. *http://code. google. com/p/honggfuzz*, 2016.

[22] Mikko Vimpari. An evaluation of free fuzzing tools. *http://jultika.oulu.fi/files/nbnfioulu-201505211594.pdf*, 2016.

[23] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[24] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328. ACM, 2017.

[25] Michal Zalewski. American fuzzy lop. *http://lcamtuf. coredump. cx/afl*, 2017.