Excel@FIT 2019
http://excel.fit.vutbr.cz

Fuzz testing of program performance

Matúš Liščinský*

Abstract
Coding new features is sometimes referred to as ”bringing new issues into the program”. And to detect these issues, especially performance issues, we often have to reach the point where ordinary inputs can never get. In this work we aim to construct automatic generator of inputs whose task will be to trigger performance fluctuations. Classical solution to automatic generation is so called fuzz testing, which is unfortunately focused on functional bugs only. So we propose to tune its rules and ways of processing the information about program run, to particularly trigger the performance bugs. We integrate our fuzzer into a performance profile manager Perun, which stores data about every run as a profile and can compare profiles of different versions. This way, we can prove that executing with certain file takes more time or memory. We tested our solution on several artificial projects which its potential when the time of program run was extended severalfold. The benefit of such solution would help developers regularly test every version of project for performance bugs and avoid them completely by automatically finding new faulty inputs.

Keywords: Performance bugs — Fuzz testing — Workload mutation — Worst-case — Algorithmic vulnerability — Denial-of-service

Supplementary Material: Project repository

*xlisci02@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Nowadays, when talking about software aspects, developers tend to focus more and more on program performance, particularly in case of mission-critical applications such as those deployed within aerospace, military, medical and financial sectors. Every project can be characterized by some time constraints such as a response to a specific action, a sampling time, or simply its runtime. When measuring performance, it is important to focus on system parameters that are significant to its performance. For example, we can collect data from the reactor every tenth of a second and if the system could not process and evaluate the data in the desired time interval, we would report a performance issue.

Performance bugs are not reported as often as *functional bugs*, because they usually do not cause crashes, hence detecting them is more difficult. Moreover, they tend to manifest with big inputs only. However, many performance patches are not that complex. So the fact that a few lines of code can significantly improve performance motivates us to pay more attention to catching performance bugs early in the development process.

Unexpected performance issues usually arise when programs are provided with inputs (often called *workloads*) that exhibit worst-case behavior. This can lead to serious project failures and even create security issues. Because, precisely composed inputs send to a program may, e.g., lead to exhaustion of computing resources *(Denial-of-Service attack)* if the input is constructed to force the worst case.

Let us assume an *unbalanced binary tree*. It is expected to consume $O(n.log(n)))$ time when inserting n elements, however, if the elements are sorted beforehand, the tree will degenerate to a linked list, and so it will take $O(n^2)$ time to insert all n elements. [1]

Manual performance testing is not a trivial process and it expects from testers awareness about used structures and logic in a tested unit. In contrast, automated testing brings a more effective way of creating test cases, which can cause unexpected performance

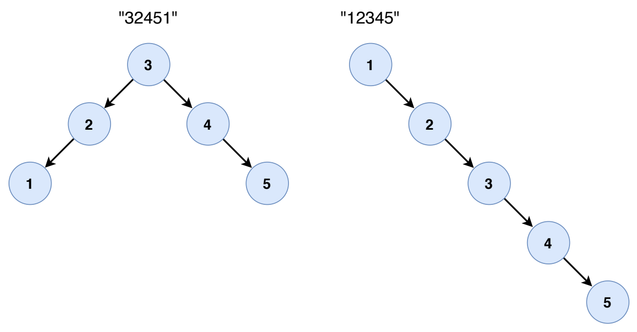

Figure 1. Unbalanced binary tree degenerating to a linked list when a sorted list is inserted

fluctuations in the target program. Unfortunately, created test cases might not detect hidden performance bugs, because it does not cover all cases of inputs. So in order to avoid this it is appropriate to adapt more advanced techniques such as the fuzzing.

Fuzzing is a testing technique used to find vulnerabilities in applications by sending garbled data as an input and then monitoring the application for crashes. Even only an aggressive random testing is impressively effective at finding faults and has enjoyed great success at discovering security-critical bugs as well. So why should not we use fuzzing to discover implementation faults affecting performance?

State-of-the-art mutational fuzzers include American Fuzzy Lop (AFL) [2] , in-process project libFuzzer [3] and many others, but these are primarily focused on finding *functional bugs*. Nevertheless, recently a performance-oriented AFL variant called PerfFuzz was proposed, which extended AFL's blind mutation strategies (flipping random bits, substituting random bytes, moving/deleting blocks of data, etc.), and sundry heuristics. [4] PerfFuzz, is a coverage-guided mutational feedback-directed fuzzing engine that uses AFL's CFG graph method and additionally creates a performance map to improve future usability estimation of tested input. Unfortunately, none of them allows to add custom mutation strategies which could be more adapted for the target program and mainly for triggering performance bugs. In addition, the mentioned tools only measure coverage of the program and do not truly measure performance, such as Perun tool provides program performance profiling.

Perun is a lightweight open-source tool which includes automated performance degradation analysis based on collected performance profiles [5]. Moreover, it offers managing performance profiles corresponding to different versions of projects, which helps user in identifying code changes that could introduce performance problems into the project's codebase or checking different code versions for subtle, long term performance degradation scenarios. Nevertheless, triggering a performance change is still highly dependent on user defined inputs.

In this work we propose new mutation strategies inspired by causes of performance bugs found in real projects and incorporating them within Perun as a new performance fuzzing technique. We believe that combining performance versioning and fuzzing could raise the ratio of successfully found performance bugs early in the process.

2. Fuzzing

Fuzzing (fuzz testing) is a form of fault injection stress testing, where a range of malformed input is fed to a software application while monitoring it for failures. [6]

The fuzzing process consists of *fuzzer* that generates input test cases either (a) using template or grammar (so called generational fuzzing) or (b) using sample workloads (so called mutational fuzzing), and *fuzzing framework* which uses generated test cases to try to trigger crashes, deadlocks or performance changes in the target application.[7]

Generational fuzzer (sometimes called grammar-based fuzzer) generates new inputs from scratch based on a template or a grammar specification. This template (e.g. protocol specification) ensures a fuzzer generates valid data for control fields such as checksums or challenge-response messages. On the other hand, creating a bulletproof template tends not to be a piece of cake and it is time-consuming.

Mutational fuzzer does not require any complex specification of input file format, just a set of sample inputs (even one single sample file suffices). New workloads are then generated by application of mutation strategies on these initial so called *seeds*.

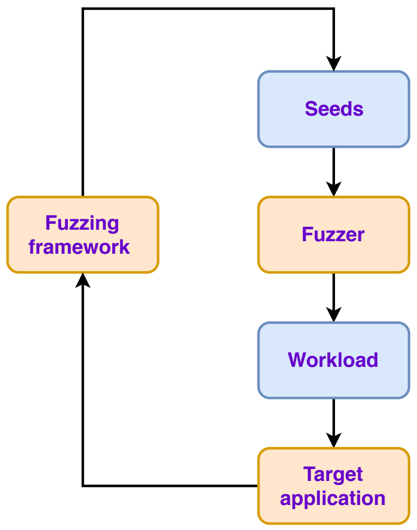

Figure 2. General scheme of mutational fuzzing.

Fuzzer chooses from a set of seeds the candidate for mutation and creates a new workload, which is tested on the target application. Framework observes its behavior and makes a decision whether this mutation is valuable and should be reused for further work or discarded. The mutation process then continues in loop either until certain number of crashes is detected or until specified timeout.

3. Performance Oriented Fuzzing

We aim to construct a lightweight *Mutation Based Fuzzing Tool* tuned for detecting performance changes, i.e. performance optimizations and degradations. An inevitable element for starting the whole process is the set of sample *seed* inputs (or workloads), called *input corpus*. The *seeds* should be valid inputs for the target application, so the application terminates on them and yields expected performance.

For different file types (or those of similar characteristics) we want to use variant groups of mutation methods. Just knowledge that *seeds* are text files, not binaries, allows *fuzzer* to avoid binary-tuned fuzz methods *(e.g random removing zero bytes, . . .)*. So, we apply domain-specific knowledge for certain types of files to trigger the performance change or find unique error more quickly.

Before running the target application with malformed inputs, it is necessary to first determine the *performance baseline*, i.e. the expected performance of the program, to which future testing results will be compared. Initial testing first measures code coverage (number of executed lines of code) while executing each initial seed. After that, median of measured coverage data is considered as baseline for coverage testing. Second, Perun is run to collected memory, trace, time or complexity resource records with initial seeds resulting into baseline profiles. Basically *performance baseline* is a profile describing performance of program on the given corpus.

The fuzzing loop itself starts with choosing one individual file from corpus using heuristic described in Section 4.3.1 This *seed* is transformed into mutations and their quantity is calculated using dynamically collected fuzz stats (Section 4.1). Every mutation file is tested with the goal to achieve maximum possible code coverage. First testing phase's importance resides in gathering the interesting inputs, which increase the number of executed lines. After gathering the interesting inputs, the fuzzer collects run-time data (memory, trace, time, complexity), transforming the data to a so called target profile and checks for performance change by comparing newly generated target profile with baseline performance profile (see [8] for more details about degradation checks). In case that performance degradation has occurred, responsible mutation file joins the corpus and therefore can be fuzzed in future to intentionally trigger more serious performance issue. The intuition is, that running coverage testing is faster than collecting performance data (since it introduces certain overhead) and by collecting performance data only newly covered paths will result into more interesting inputs.

Listing 1. Pseudocode of Performance Fuzzing Algorithm

```
results = []
base_cov = init_cov_test(corpus)
base_profile = init_perun_test(corpus)
mutation_rules = choose_rules(corpus)
# Fuzzing loop
while timeout not reached:
  interesting inputs = []
  # Coverage-guided testing
  while executions_limit not reached and
  collected_files_limit not reached:
    candidate = choose_parent(corpus)
    mutations = fuzz(candidate,
        mutation_rules, fuzz_stats)
    # Gathering interesting mutations
    interesting_inputs += test_for_cov(
        mutations, base_cov, incr_ratio)
    corpus += interesting_inputs
    update_stats(fuzz_stats)
  # Profile-guided testing
  results += test_with_perun(
      interesting_inputs, base_profile)
  update_stats(fuzz_stats)
```

Implemented line coverage collection is possible only in the presence of source files. Nevertheless, our fuzzer counts on such a situation and provides fuzz testing even without them. Skipping the first testing phase fuzzing becomes blinder and less controlled because of not filtering probably not usable test cases and degradation has to be caught just after one single application of a fuzz method.

4. Fuzzer Engine Implementation

4.1 Mutation Methods Selection

Fuzzer distinguishes between text and binary files and for each defines a set of basic mutation strategies. Further, fuzzer can be extended by other strategies based on file mime-type. We select corresponding strategies on the beginning, based on first loaded input file.

We propose new mutation techniques inspired by real projects performance bugs[1]. Suppose the seed input is the string ”the quick brown fox jumps over

[1] See https://accidentallyquadratic.tumblr.com for more info

the lazy dog". We propose for text files the following mutation strategies, each one listed with example of the mutation results:

(1) *double the size of a line:*
"the quick brown fox jumps over the lazy dogthe quick brown fox jumps over the lazy dog"

(2) *remove whitespaces of a line:*
"thequickbrownfoxjumpsoverthelazydog"

(3) *sort words or numbers of a line:*
"brown dog fox jumps lazy over quick the the"

(4) *repeat random word of a line:*
"the quick brown fox jumps over the lazy dog dog dog dog dog dog"

(5) *repeat whitespaces:*
"the   quick brown fox jumps over the lazy dog"

(6) *remove random character, line, or word:*
"the quick brown fx jumps over the lazy dog"
"the quick brown jumps over the lazy dog"

(7) *prepend or append whitespaces to a line*
"   the quick brown jumps over the lazy dog   "

For example, the intuition behind the mutation rule (4) is mainly based on structure of hash tables. If the analyzed program internally stores words in hash tables, then repeated lookup in hash table may induce degradation.

For binary files we use classical mutation strategies, that are, e.g., used in AFL. Suppose binary file with content: "This is !binary! file.\0". We can then use the following mutation rules:

(1) *remove or add random zero bytes (for C strings):*
"This is !binary! file."
"This is !\0binary! file.\0"

(2) *remove or add random byte:*
"This is !inary! file.\0"
"This is !binar$y! file.\0"

(3) *swap random bite or byte:*
"This is %binary! file.\0"
"This is !binary! f&le.\0"

We cannot really apply domain-specific knowledge for fuzzing binary files in order to trigger performance changes more often. However, in PerfFuzz [4], authors show that even with binary mutations one can achieve degradations quite often.

At last, we will conclude with several mime-type-specific mutation rules such as:

(1) *remove an attribute:*
< book id="bk106" >

(2) *remove only attribute name or value*
< book id="bk106" "457" >
< book id="bk106" pages="" >

(3) *add or remove the tag*
< book id="bk106" pages="457" ><library>

in sample line < book id="bk106" pages="457" > of XML file.

Currently we limit fuzzer to work with one file type according to the input, however fuzzing of multiple file types is our future work.

4.2 Initial testing

Baseline results are essential for detecting performance changes, because newly mutated results have to be compared against some expected behaviour or value. Initial seeds becomes test cases and are used to collect performance baselines. By default, our initial testing as well as testing in fuzzing loop (Section 4.3) interleaves two phases described in more details below.

4.2.1 Coverage-guided testing

For achieving better results in triggering performance changes it is generally recommended to monitor the code coverage during the testing. The intuition is that by monitoring how many paths are covered and how often they are executed, we can more likely encounter a performance bug. In our fuzzer, we use Gcov tool to measure code coverage information of a program.[2] Total count of executed code lines through all source files represents performance indicator for this phase. An increase of the value means that more instructions have been executed, (for example, some loop has been repeated more times) so we hope that performance degradation will be triggered as well. Note that the limitation of this approach is that it does not track uniquely covered paths, which could trigger performance change as well.

4.2.2 Profile-guided testing

Much of the work presented in this section has been already implemented within Perun tool. Perun runs the target application with a given workload, collects performance data about the run such as runtime or consumed memory, and stores them as a persistent so called profile, i.e. the set of performance records.

[2] Gcov tool - https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Analogically to the previous section, we need a performance baseline, with which we will compare newly generated results. Profiles measured on fuzzed workloads (*target profiles*) are then compared with a profile describing performance of program on the initial corpus (*baseline profile*). In order to compare the pair of baseline and target profiles we use sets of calculated regression models, which represents the performance using mathematical functions computed by least-squares method. From both of these sets for each function we select its models with the highest value of *coefficient of determination* R^2, which represents how well the model fits the data, as well as its corresponding linear models. For both pairs of best models and linear models we compute a set of data points by simple subtraction of these models. Then we use regression analysis to obtain a set of models for these subtracted data points. Moreover, for the first set of data points, corresponding to best-fit models, we compute the relative error, which serves as a pretty accurate check of performance change. All of these regressed models are then given to the concrete classify functions, which gradually returns detected degradation for each function. [8]

4.3 Fuzzing loop

In this section, we described the main loop of whole fuzzing process and some of its most significant parts.

4.3.1 Parent input selection

Initially, input corpus is filled with seeds, which will be parents to newly generated mutations (we can call them *parent inputs*). During the fuzz testing, successful mutations join this corpus and become *parent inputs* too. The success of the input is represented by the *fitness score*, which is a numeric value indicating workload's point rating. The total score is calculated by the fitness function as the sum of two partial rates:

1. **Increase coverage rate**: The value indicates how much coverage changes if we run the program with the input, compared to base coverage measured for initial corpus. Basically, it is a ratio between coverage measured with the input and base coverage:
$$icovr_{\text{input}} = cov_{\text{input}}/cov_{\text{base}}.$$
2. **Performance change rate**: In general, we compare the newly created profile with that baseline profile (for details see Section 4.2.2) and the result is a list of located performance changes (n particular *degradations*, *optimizations* and *no changes*). Performance change rate is then computed as ratio of degradation in the result list:
$pcr_{\text{input}} = \text{cnt}(degradation, result)/\text{len}(result)$

All *parents* are kept sorted by their scores, and selection for further mutation consists of dividing the seeds into five intervals such that the seeds with similar value are grouped together. At first, we assign weight to each interval using linear distribution. Right after that, we perform a weighted random choice of interval. Finally we randomly choose a parent from this interval, whereas differences between parent's scores in the same interval are not very notable.

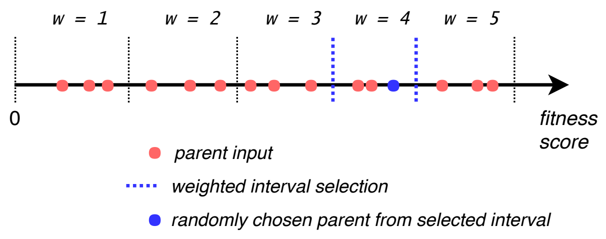
w = 1
w = 2
w = 3
w = 4
w = 5
0
fitness score
parent input
weighted interval selection
randomly chosen parent from selected interval

Figure 3. Parents are divided to intervals according to their fitness score. Weighted choice of interval determines the chunk of seeds, from which final candidate is randomly chosen.

4.3.2 Fuzzing

After we compute baseline data for input process and chose appropriate mutation rules, we can use fuzzer to gradually apply them and generate new workloads. However, it is necessary to determine how many files (N) to generate in each iteration of fuzzing loop. We dynamically calculate the value of N according to the fuzz statistics of rules during the fuzzing process. Statistical value of rule f is a function:
$$stats_f = (degs_f + icovr_f)$$
where $degs_f$ represents the number of occurred degradation by applying rule f, and $icovr_f$ stands for how many times coverage was increased applying rule f.

Fuzzer then calculates the number of new mutations for every rule to be applied in four possible ways:

1. $N = 1$, the fuzzer will generate one mutation per rule. The simple solution without the impact of dynamic collected statistical data and therefore all the rules are equivalent.
2. $N = stats_f + 1$, the fuzzer will generate mutations appropriately to the statistical value. More mutation workloads are generated, in case that rule f has not caused a change in coverage or performance (i.e $stat_f = 0$) yet, the same result as in first strategy.
3. Depends on $total$, which stands for total number of degradation or coverage increases. The ratio between $stats_f$ and $total$ determines the value of $prob_f$, i.e. the probability whether the rule f should be applied.

$$prob_f = \begin{cases} 1 & \text{if } total = 0 \\ 0.1 & \text{if } stats_f/total < 0.1 \\ stats_f/total & \text{otherwise} \end{cases}$$

and N then:

$$N = \begin{cases} 1 & \text{if } random <= prob_f \\ 0 & \text{otherwise} \end{cases}$$

Until some change in coverage occurs, (i.e. when $total = 0$), one new workload to generate is assigned to each rule. After several iterations, more successful rules have a higher probability, so they are applied more often. Rules with very poor ratio would be highly ignored. However, since they still may trigger some changes we round them to probability of 10%.

4. Modified third strategy combined with the second one. When the probability is high enough, that the rule should be applied, the amount of generated workloads is appropriate to the statistical value. Probability $prob_f$ is calculated equally, the equation for N is modified to :

$$N = \begin{cases} stats_f + 1 & \text{if } random <= prob_f \\ 0 & \text{otherwise} \end{cases}$$

Our fuzzer uses this method by default, because it guarantees that it will generate enough new workloads and will filter not successful rules without totally discarding them.

4.3.3 Gathering interesting mutations

We usually run fuzzing for a longer period of time trying to trigger as many changes or faults as possible. To maximize the number of found changes we try to avoid running the target application with workloads with a poor chance to succeed.

In the situation, when the workload does not exceed the coverage threshold, it is not significant for us, because instruction path length is not satisfactory, hence we discard this workload. The threshold is multiple of base coverage, set to 1.5 by default, but it can also be specified by the user. During the testing, fuzzed workload can cause that target program terminates with an error (e.g SIGSEGV, SIGBUS, SIGILL, ...) or it hangs (runs too long). Even though we are not primarily focused on faults, they are interesting for us too because an incorrect internal program state can contain some degradation and in error handlers can also be hidden degradation.

4.3.4 Performance changes detection

All the interested mutations are now collected and ready for real testing revealing performance changes. Testing is done similarly to initial profile-guided testing (Section 4.2.2), but instead we test with fuzzed interesting inputs. If performance degradation occurs, we rate and assign the score to the workload using fitness function and classify the result as the final result causing performance degradation.

5. Experimental Evaluation

We tested our fuzzing machine on artificial examples to measure its efficiency of generating worst-case mutations. We mentioned in the introduction that time consumption of inserting to unbalanced binary tree (UBT) highly depends on the order of insertion. We randomly generated 1 000 numbers in the range of <0,1 000> and used them as an initial seed to a program that create an UBT and then iteratively inserts elements. We set the maximum size of generated mutation to 8800 bytes to avoid large input files. In first minute of fuzzing loop, we already reported 60 files which reported performance degradation. Analysis of worst-case mutations confirmed, that unbalanced binary tree degenerates to a linked list when a sorted list is inserted. Moreover, when a fuzzer maximizes the size of mutation, which is only around 2.27 times larger than the original seed, average time spent by the target program grows rapidly.

We further tested standard library list `std::list`[3], which is usually implemented as a doubly-linked list, and performed a search with `std::find`.[4] The tested program reads strings from a file, saves them to list and subsequently performs a search for each of them. Initial seed contains 10 000 random words, and using a rule that sorts the words led to 14 times longer execution time of the program. For comparison, the same mutation but with randomly shuffled words causes the degradation as well, however, execution time was only 2.53 times worse than expected behavior and is probably caused by the necessity of parsing more words.

Table 1. Worst-case mutations as input to target programs working with selected data structures, and their impact on performance.

structure	time degradation	cov increase
UBT	**9.28x**	**41.56x**
std::list + std::find	**14.01x**	**15.05x**

Table 1 shows worst-case generated mutations, their time and coverage characteristics in comparison

[3] std::list - https://en.cppreference.com/w/cpp/container/list
[4] std::find - https://en.cppreference.com/w/cpp/algorithm/find

with an initial seed. You can see that time-consumption increased more than *9 times*, respectively *14 times*, when the programs were provided with the worst-case inputs and that it has been executed severalfold more lines of code.

In our second experiment, we tested artificial programs which use `std::regex_search`[5], on the several regular expressions inspired by ReDoS attacks[6] (regular expression denial of service). First is the simplified regular expression that caused outage of StackOverflow in July, 2016. The regex tries to match whitespaces at the end of a line, but is weak with possible backtracking. The other two regular expressions have a similar problem, when carefully constructed inputs can cause catastrophic backtracking and are therefore considered to be vulnerability.[7]

Table 2. Worst-case generated inputs for specific regular expressions with their impact on performance in comparison with an initial seed.

regex	time degradation	coverage increase
\s+$	5.79x	15.59x
^(.*?,){10}P	2 897.23x	2 442.37x
**	940.44x	10 873.94x

** <html>.*?<head>.*?</head>.*?<body[^>]*>.*?</body>.*?</html>

Analysis of worst-case mutations showed, that in the first case, long sequences of whitespaces not ending with the end of line caused the regex engine to backtrack repeatedly. The second regular expression is built for CSV (Comma-separated values) files, and tries to find lines where the 11th item on a line started with a 'P'. At the point when the 11th field does not start with a 'P', the engine will backtrack as well. At last, a simple regular expression matches a complete HTML file, however, when a file was cut at the end and does not contain valid ending html tags, the engine takes too much time as well.

6. Conclusions

In this paper, we introduced a fuzzing machine generating malicious inputs focusing on performance weaknesses. We use specific methods to mutate the files, dynamically analyze their efficiency, collect user coverage information and use the Perun tool [5] to measure information of program run.

Our future work will focus mainly on effective visualization of fuzzing results, to illustrate which particular inputs challenge the program, what changes makes the program consume more resources, or how many changes we can detect in time. Moreover, we want to add new domain-specific mutation methods, work on support for fuzzing with multiple file types, and also improve parent rating and selection by deeper analysis of program run. At last, we plan to evaluate our solution on real-world projects and potentially report new unique performance bugs.

[5] https://en.cppreference.com/w/cpp/regex/regex_search
[6] https://en.wikipedia.org/wiki/ReDoS
[7] https://www.regular-expressions.info/catastrophic.html

Acknowledgements

I would like to thank for the support received from Red Hat company, my supervisors and colleagues from VeriFIT performance team — Tomáš Fiedor, Adam Rogalewicz, Hana Pluháčková, Tomáš Vojnar, Šimon Stupinský and Jiří Pavela.

References

[1] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association.

[2] Michał Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.

[3] libfuzzer – a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`.

[4] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, New York, USA, 2018. ACM.

[5] Perun: Performance version system. `https://github.com/tfiedor/perun`.

[6] Toby Clarke. Fuzzing for software vulnerability discovery. Technical Report RHUL-MA-2009-04, Department of Mathematics, Royal Holloway, University of London, Egham, Surrey TW20 0EX, England, February 2009.

[7] Emil Edholm and David Göransson. Escaping the fuzz - evaluating fuzzing techniques and fooling them with anti-fuzzing. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2016.

[8] Pavela Jiří and Šimon Stupinský. *Towards the detection of performance degradation*. In *Excel@FIT'18*.