BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

http://fit.vut.cz

Project practice
2019/2020

Analysis, Modeling and Prediction of Program Performance Based On Recent Testing Techniques

Matúš Liščinský*

Abstract

In general, the term *software testing* often focuses on the idea of functional testing and searching for functional deficiencies. On the other hand, a functionally well-prepared project that does not meet a certain level of performance loses its justification and market position. Therefore, *performance testing* should also have a place in development. This problem seems to be perfectly solved by Perun, an automatic performance tester with the ability to track project history. Perun already includes the unit providing fuzz testing described in the paper *Fuzz testing of program performance* [12] from which some sections of this text are taken. Fuzzing tool works on the basis of collecting information about code coverage, which it aggregates into one value (number of executed lines of code). With this approximation, the analysis loses accuracy and weakens the ability to detect a significant but local change in code coverage. The aim of this work is to minimise these shortcomings by improving the analysis of coverage when running a program with mutated inputs. The main pillar of improving the analysis is the static call graph of the target program. Coverage data are assigned to the call graph, what puts them in a certain reasonable structure. In addition, this structure stores on two types of coverage data: inclusive and exclusive. In addition, the call graph is divided into individual paths from the root node to the leaf nodes. We do not work with only one value as before, but directly with the vectors of inclusive/exclusive coverage of paths, which are compared across the run of the tested program with malformed inputs. This way we can do a better analysis of the mutated inputs and therefore a better decision whether a mutation is suitable for further testing by one of the selected Perun collectors (time, memory, trace).

Keywords: Fuzzing — Performance — Perun — Mutation — Coverage — Inclusive Coverage — Exclusive Coverage — Call Graph — Unique Paths — Worst-case

Supplementary Material: Perun repository

*xlisci02@stud.fit.vut.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 characterised 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.

Performance bugs are not reported as often as *functional bugs*, because they usually do not cause crashes, hence detecting them is more difficult. 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 behaviour. 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 $O(n \cdot log\, n)$ 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. [7]

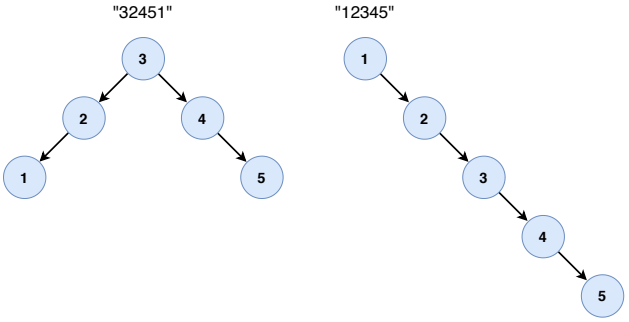

Figure 1. Unbalanced binary tree degenerating to a linked list when a sorted list is inserted

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 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.

State-of-the-art mutational fuzzers include American Fuzzy Lop (AFL) [13], in-process project libFuzzer [4] 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 [10]. 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 [8].

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 quite 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*.

Fuzzer chooses from a set of seeds the candidate

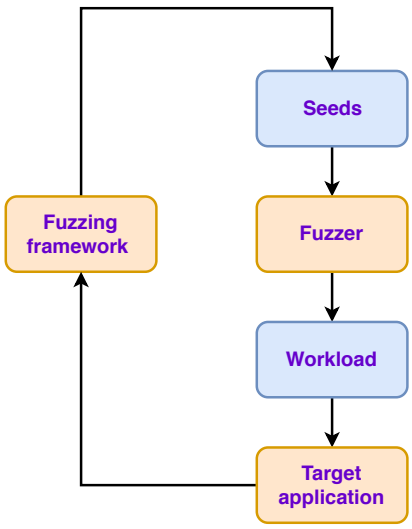

Figure 2. General scheme of mutational fuzzing.

for mutation and creates a new workload, which is tested on the target application. Framework observes its behaviour 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

This work is based on the existing Perun extension providing mutation based fuzz testing, and tuned for detecting performance changes [11]. Therefore, it would be advisable to mention at least in highlights how is this tool built and discuss the main pillars it is standing on. The previously proposed tool for fuzzing operated on the mutation based approach, so 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.

To trigger a performance change of the target system, we systematically launch it with the differently malformed inputs. The inputs are actually the same as in the input corpus, but several times transformed or modified by the sundry mutation rules.

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. This *seed* is transformed into mutations and their quantity is calculated using dynamically collected fuzz stats. 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 [9] 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.

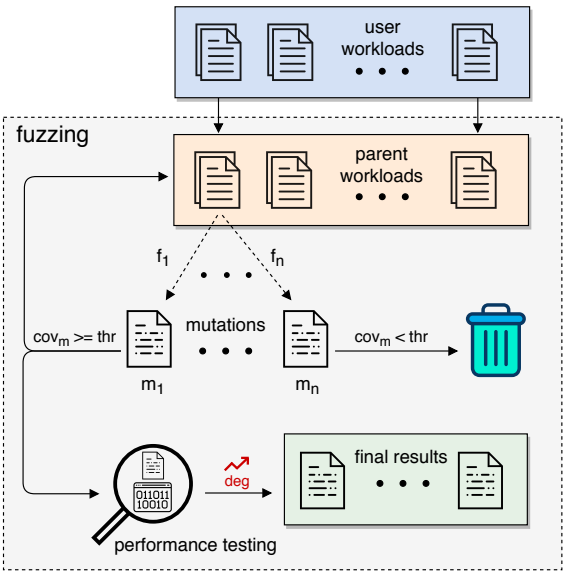

Figure 3. Lifetime of workloads [11].

4. Improving the coverage analysis

Unfortunately, the described principle of operation of the fuzz testing tool still has its pitfalls. One of them is that when choosing the so-called interesting mutations in coverage testing, there is a significant approximation. The sum of executed LOCs in the run is well used especially for smaller projects, with a smaller number

of LOCs, but with more robust programs with modules containing several thousand LOCs, the local maximum coverage can be easily lost and the desired change is not detected.

4.1 Engaging call graph

This work aims to improve the current analysis of workloads to increase its general ability to detect significant changes in code coverage. We have therefore decided to add to our purely dynamic analysis a static element: call graph. The call graph is an oriented graph, which simply describes the mutual relationship of calls of individual program subroutines. Each node represents a specific subroutine, and each edge (f, g) indicates the call of subroutine g from within subroutine f. The information about code coverage and static call graph together builds the idea of the creation of an annotated call graph, on which this analysis is based.

Each node of the call graph does not only carry information about which function it represents but also its inclusive and exclusive coverage. In our case, inclusive coverage means how many lines were executed in a given function during the program run. The exclusive coverage variant represents the number of calls of this function. One run of the program will not differ from another by the structure of the call graph, but by the information stored in its nodes. Unlike the original approach, where the coverage indicator was a single number, we now work with multiple data stored in the graph structure, which gives us the chance to more accurately detect and locate eventual change in code coverage.

4.2 Use of call graph for coverage analysis

Incorporating the new approach of analysing the code coverage will affect other components of the fuzzing tool that we will have to deal with. The key component is decisive whether the given mutation is interesting for us and will the analysis be performed with it by the Perun tool. In the previous principle of analysis, the success of the mutation was evaluated according to whether it was able to force higher coverage than its parent and at the same time its rate of increase in coverage over baseline coverage calculated on the input corpus. In the renewed analysis, we work with an annotated call graph, in which we find all unique paths from the root to the end node. Each such path is a separate entity with two types of information: inclusive and exclusive coverage. Along the way, we define inclusive, resp. exclusive coverage of the path as the sum of inclusive, resp. exclusive coverage for individual functions all the path. The resulting indicators of coverage are the vectors of inclusive exclusive coverage of all such paths. This allows us to better locate the change and monitor source code coverage with less granularity.

4.3 Recursion in call graph

We yield information about a target project call graph from an `angr` [2] binary analysing framework and use them to build our representation of the call graph. Besides building we dynamically create and store all paths from the root to the leaf node. However, not every path must contain a leaf node, which brings us to the problem of cycles in the graph. From the description of a call graph, it is clear that the cycle in the graph indicates recursive procedure calls. This can happen when a function calls another function that (directly or indirectly) calls the original function.

As is described in Section 4.2 we gather the coverage information of every node (procedure) in the path to define coverage performance indicator of mutation. To prevent us from cycling between nodes in case of recursion, we decided to cut the path whenever a cycle is detected, i.e. at the moment, we get to the node twice in the context of currently building path. This node will not be included in the path, because we would involve the same information twice in one path. Example of a call graph containing cycles one can see in Figure 4.

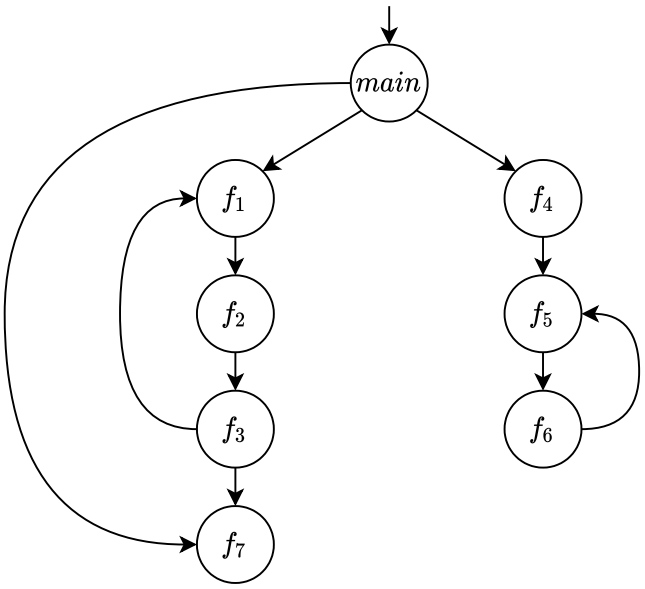

Figure 4. Example of call graph demonstrating extracting the paths in recursion program.

The paths we construct would be as follows:

- $main \to f_7$
- $main \to f_1 \to f_2 \to f_3$
- $main \to f_1 \to f_2 \to f_3 \to f_7$
- $main \to f_4 \to f_5 \to f_6$

5. Impact on the existing solution

The described analysis brings up the necessity of modifying the following three important components of fuzzing which uses coverage analysis:

1. determining of baseline coverage
2. determining whether the mutation is in our interest (whether it triggered sufficient coverage increase)
3. determining the mutation score if it caused a performance change

5.1 Baseline coverage

One of the required arguments for a fuzzing tool is the input corpus which determines the baseline coverage. For all n seeds from the corpus, the program runs once and the result is n vector pairs (*inclusive*, *exclusive*). Baseline coverage again consists of a pair of vectors (*inclusive*, *exclusive*), where each element cov_i of any vector is calculated as the median of the values of all inclusive, resp. exclusive vectors on the same index (i.e. the median of the inclusive or exclusive coverage within an individual path). We can describe this process informally as follows:

$baseline = (inclusive, exclusive)$
$inclusive = (path_0.cov\_inc,\ path_1.cov\_inc,\ \ldots)$
$exclusive = (path_0.cov\_exc,\ path_1.cov\_exc,\ \ldots)$
$path_i.cov\_inc = median(seed.path_i.cov\_inc \mid \text{for each } seed \in corpus)$
$path_i.cov\_exc = median(seed.path_i.cov\_exc \mid \text{for each } seed \in corpus)$

5.2 Mutation filtering

Filtering mutations based on their ability to cause an increase in coverage is an important pre-preparation before testing an input with one of the Perun collectors. Therefore, we paid the most attention to it and tried several different approaches on how to decide this problem. From previous experience with fuzzing, the approach of comparing run coverage with a given mutation with baseline coverage and with run coverage with its parent mutation (or seed) proved to be effective. Methods of comparison of individual vectors as coverage indicators for determining whether we keep the mutation are presented below:

mutation *mut* is interesting $\Leftrightarrow$

1. $\exists path \in cg.paths:$
   $path.cov\_inc > parent.path.cov\_inc \wedge$
   $path.cov\_inc > th * baseline.path.cov\_inc$
   $\wedge\ \exists path \in cg.paths:$
   $path.cov\_exc > parent.path.cov\_exc \wedge$
   $path.cov\_exc > th * baseline.path.cov\_exc$

2. $\exists path \in cg.paths:$
   $path.cov\_inc > parent.path.cov\_inc \wedge$
   $path.cov\_inc > th * baseline.path.cov\_inc$
   $\vee\ \exists path \in cg.paths:$
   $path.cov\_exc > parent.path.cov\_exc \wedge$
   $path.cov\_exc > th * baseline.path.cov\_exc$

3. $\forall path \in cg.paths:$
   $path.cov\_inc > parent.path.cov\_inc \wedge$
   $path.cov\_inc > th * baseline.path.cov\_inc$
   $\wedge\ \forall path \in cg.paths:$
   $path.cov\_exc > parent.path.cov\_exc \wedge$
   $path.cov\_exc > th * baseline.path.cov\_exc$

4. $\forall path \in cg.paths:$
   $path.cov\_inc > parent.path.cov\_inc \wedge$
   $path.cov\_inc > th * baseline.path.cov\_inc$
   $\vee\ \forall path \in cg.paths:$
   $path.cov\_exc > parent.path.cov\_exc \wedge$
   $path.cov\_exc > th * baseline.path.cov\_exc$

5. $avg(mut.inclusive/parent.inclusive) > 1 \wedge$
   $avg(mut.inclusive/baseline.inclusive) > th \wedge$
   $avg(mut.exclusive/parent.exclusive) > 1 \wedge$
   $avg(mut.exclusive/baseline.exclusive) > th$

where *cg* stands for call graph of the target application, *th* represents the dynamically computed ratio threshold, and *avg* is a function that computes an average value. Experiments on artificial projects have shown, that method number three is too strict because it requires a change in both types of coverage on all paths, and for paths that could not be affected by the input, the desired change will never be achievable. Similarly, method number four requires sufficient change for all paths in at least one type of coverage, which has shown to be a better method but still not sufficient.

The second method, which poses the least requirements for increasing coverage, proved to be the best solution for capturing the initial degradation. Finding the first mutation often forms a stepping stone in the search for worst-case behaviour inputs. If we would like to describe this condition verbally, then the mutation is marked as interesting only and only if there is sufficient inclusive or exclusive coverage growth on any path. First and the last method has been acting in nearly the same way because they reached similar results. However, our intuition is that the second method will be more suitable and general especially in case of real-world projects. All the methods are implemented anyway, so the user can choose a method by himself.

5.3 Parent score

If the Perun analysis shows that target application run with a mutated input triggers a performance degradation, we join the mutation to the parent set. Every parent has to be rated in order to choose the best parents in the process of input mutation. Rate of the parent should reflect the success it reached when it was executed with the target application, so in the previous solution, we compute new parent score as a ratio of reached coverage and baseline coverage. We work with a pair of vectors which identifies the baseline coverage and another pair of vectors representing inclusive and exclusive coverage of paths within run with the mutation. Inside the operation of computing new score, we firstly get a pair of change vectors, which are simply an element-wise ratio between coverage of mutation and baseline (both inclusive and exclusive). The resulting score is then defined as an average coverage ratio of all paths and types of coverage, so the average value of the change vectors. In the end, this score is multiplied by $(1 + deg\_ratio)$, to reflect the result from the Perun analysis, as well as in the original approach.

6. Evaluation of the new approach

We subjected the renewed method to several tests to determine its behaviour compared to the original approach. The evaluation included five tests in total, each consisting of fuzzing a certain project for 300 seconds. Each of the projects is artificial with a nested performance bug. More about the tested projects one can read in the evaluation part of the original solution text [11].

The aim was to determine the competitiveness of the new method of working with coverage and to test its use in practice. The results can be viewed in Table 1.

Project	Old approach		New approach	
	cov ratio	hangs	cov ratio	hangs
(email+class) rEx	5915.44	0	7.02	4
std::find, std::list	1.54	0	1.18	0
java hash function	3.58	0	3.56	0
stackoverflow rEx	8.45	0	4.57	0
ubt	6.49	0	5.72	0

Table 1. Evaluation of new approach on the relatively small artificial projects. Less important data in the table are marked with grey colour. Column *cov ratio* represents coverage ratio between the best-rated mutation and baseline coverage, and column *hangs* shows how many mutations that exceed the hang timeout (in this case 30 seconds) we found during fuzzing.

The first test-case `(email+class)rEx` in the evaluation Table 1, which is the largest of those tested, indicates that we achieve better results as we found up to four hang-causing inputs. Note that *cov ratio* is an insignificant value in this case, because only the best-mutated input excluding hang-causing and error-causing mutations is taken into this statistic. This is also the only tested project where we found an enormous slowdown leading to a hang state.

The second largest test-case is a project called `java hash function`, where, as we can see only a minimal difference between the *cov ratio* values, as well as in the test-case `std::find, std::list`. One of the most interesting was a test program containing a vulnerable regular expression previously used by the StackOverflow domain. With the new approach, we were only able to obtain a mutation that increases coverage by about half compared to the mutation from testing with the original method. It turned out that this phenomenon was caused only by a short time of fuzz testing because this behaviour is caused by the deeper coverage analysis and more complex work with the collected coverage data.

Later, repeated fuzzing with the same parameters and runtime 3600 seconds (1 hour) managed to find a mutated input which forced the *cov ratio* value up to 30.34, what means over 30 times higher coverage in comparison to the baseline value.

The last tested project was `ubt`, that stands for Unbalanced Binary Tree. At first look at this row of the table, it is clear that with the original approach we achieved a better result because the best mutation achieved a higher decisive value *cov ratio*. However, analyses with the Perun time collector showed that none of the mutations found, even the best rated one, caused a sufficient performance change to allow the input to be classified as degrading. On the other hand, with the seemingly worse result of the new method, up to 2 malformed inputs were found, which forced the program to run long enough to be marked as causing a performance change after Perun analysis.

Realising that we used relatively small projects for evaluation, we achieved solid results. Our analysis should have the potential and show its strengths mainly in fuzz testing of larger projects. However, evaluation and testing on projects with more massive amounts of code, where the real-world projects also fall is a task for future work.

7. Visualisation of input impact on the program paths.

Since we brought a new perspective to the fuzzing process on the analysis of coverage working with a larger volume of data, we decided to use it and provide it to the user in a suitable form. Thanks to the analysis of the call graph and the coverage of individual paths, we are also able to store information about the coverage of individual paths. After the fuzzing, we provide a list of the most influential paths, and their maximum increase in coverage, which we managed to achieve during the fuzzing. This data are also visualised in the form of two heat maps, illustrating the maximum achieved an increase in inclusive/exclusive path coverage, as one can see in the Figures 5 and 6.

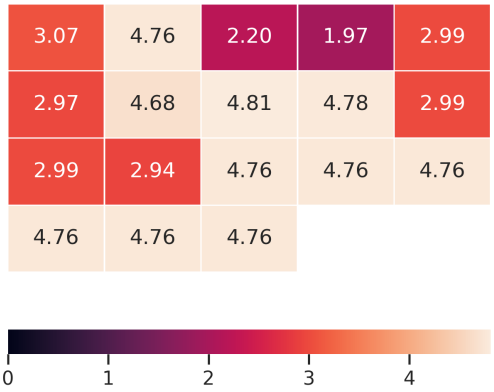

Figure 5. The resulting heat map showing the maximum increase ratio in the **inclusive** coverage of the artificial project paths. As one can notice some paths reached the same max coverage ratio with baseline coverage. this phenomenon usually occurs when two functions have identical prefix except for the leaf function which is from the system headers files. Gathering coverage information ensures `gcov` tool, and this tool does not include inclusive coverage data of functions inside the system libraries, therefore they are not present in their summaries either [1]. Usually, within testing custom projects inclusive coverage of these built-in functions is not interesting, hence we ignore it for now.

The initial intuition to overcome this problem tends to merge such paths. One path would wrap several mutually similar routes with common inclusive coverage. However, these paths may differ in their exclusive coverage, which we try to obtain using a cross-reference table, as explained in the description of Figure 6. Therefore, we cannot combine such paths into one, but take each as a separate entity.

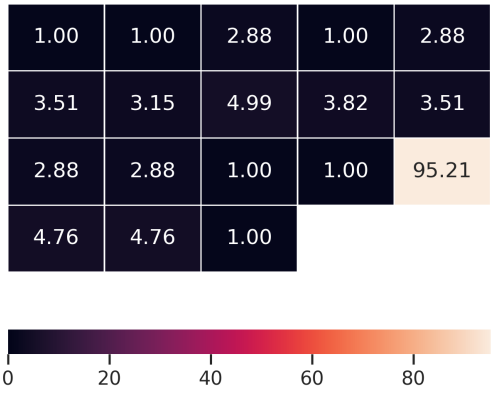

Figure 6. The resulting heat map showing the maximum increase in the **exclusive** coverage of the artificial project paths. The heat map is the result of the same fuzz testing as the heat map of inclusive coverage ratios in Figure 5. Again, we observe similarity among the ratios of the paths. Note that in contrast to the inclusive type of coverage, we provide exclusive coverage even of the system library functions thanks to the cross-reference table obtained by the `cflow` utility [3]. In exclusive coverage, we take into account only function calls, so in case of paths where the prefix is the same except for the last function, there is a high probability that the last functions of these paths will be called the same or very similar number of times. Let us assume two paths:
$main \rightarrow f \rightarrow g \rightarrow malloc$
$main \rightarrow f \rightarrow g \rightarrow free$.
When working properly with memory, the $malloc$ and $free$ functions are supposed to be called the same number of times, that implies their exclusive coverage equals. The same situation can occur in case of the leaf functions that provided input do not affect at all.

8. Conclusions

In this paper, we introduced a new approach of deeper coverage analysis to improve mutation selection of fuzz testing, implemented within Perun tool [5]. We use static call graph annotated with the inclusive and exclusive coverage of the procedures, i.e. nodes of the call graph. Using this principle, we aim to detect a local maximum in coverage, not visible in the original approach. Promising results from evaluation proves, that we sufficiently reinforce the ability to reveal performance fluctuations using the renewed coverage analysis.

Our future work will focus mainly to perfect the fuzzing tool by adding new domain-specific mutation methods, work on support for fuzzing with multiple

file types, and also trying to improve the analysis by observing the loops coverage, static source file analysis or dynamic derivation of fuzzing parameters on the fly. At last, we plan to evaluate our solution on real-world projects and potentially report new unique performance bugs.

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] *gcov — a Test Coverage Program*. [Online; visited 24.6.2020]. Available at: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[2] *angr — Python framework for analysing binaries*. [Online; visited 24.6.2020]. Available at: https://angr.io/.

[3] *GNU cflow*. [Online; visited 24.6.2020]. Available at: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[4] *LibFuzzer – a library for coverage-guided fuzz testing*. [Online; visited 24.6.2020]. Available at: https://llvm.org/docs/LibFuzzer.html.

[5] *Perun: Performance Version System*. GitHub. [Online; visited 24.6.2020]. Available at: https://github.com/tfiedor/perun.

[6] CLARKE, T. *Fuzzing for software vulnerability discovery*. RHUL-MA-2009-04. Egham, Surrey TW20 0EX, England: Department of Mathematics, Royal Holloway, University of London, February 2009. Available at: https://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-04.pdf.

[7] CROSBY, S. A. and WALLACH, D. S. Denial of Service via Algorithmic Complexity Attacks. In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. Berkeley, CA, USA: USENIX Association, 2003. SSYM'03. Available at: http://dl.acm.org/citation.cfm?id=1251353.1251356.

[8] EDHOLM, E. and GÖRANSSON, D. *Escaping the Fuzz - Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing*. 2016. Master's thesis. Department of Computer Science and Engineering, Chalmers University of Technology.

[9] JIŘÍ, P. and STUPINSKÝ Šimon. *Towards the detection of performance degradation*. In: *Excel@FIT'18*.

[10] LEMIEUX, C., PADHYE, R., SEN, K. and SONG, D. PerfFuzz: Automatically Generating Pathological Inputs. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, USA: ACM, 2018. ISSTA 2018. Available at: http://doi.acm.org/10.1145/3213846.3213874. ISBN 978-1-4503-5699-2.

[11] LIŠČINSKÝ, M. *Fuzz testing of program performance*. Brno, CZ, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[12] LIŠČINSKÝ, M. *Fuzz testing of program performance*. In: *Excel@FIT'19*.

[13] MICHAŁ ZALEWSKI. *American Fuzzy Lop*. Available at: http://lcamtuf.coredump.cx/afl/.