BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FUZZ TESTING OF PROGRAM PERFORMANCE
FUZZ TESTOVÁNÍ VÝKONU PROGRAMU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	MATÚŠ LIŠČINSKÝ
SUPERVISOR VEDOUCÍ PRÁCE	Doc. Mgr. ADAM ROGALEWICZ, Ph.D.

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

19090

Student:	**Liščinský Matúš**
Program:	Informační technologie
Název:	**Fuzz testování výkonu programu**
	Fuzz Testing of Program Performance
Kategorie:	Algoritmy a datové struktury

Zadání:

1. Seznamte se s projektem Perun (správcem výkonnostních profilů) a s metodami profilováním programů.
2. Prostudujte techniku fuzz testování se zaměřením na testování výkonu či odhalování výkonnostních chyb. Seznamte se s existujícími fuzz testery (AFL, PerfFuzz, atd.).
3. Navrhněte a implementujte modul pro fuzz testování se zaměřením na testování výkonu aplikací v rámci projektu Perun.
4. Navrhněte a implementujte modul pro interpretaci výsledků výkonnostních testování získané fuzz testováním.
5. Demonstrujte řešení na alespoň 3 případových studiích.

Literatura:

- Caroline Lemieux, Rohan Padhye, Koushik Sen, Dawn Song.: PerfFuzz: Automatically Generating Pathological Inputs
- Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, Renwei Zhang.: Fuzz testing in practice: Obstacles and solutions
- Oficiální stránky projektu AFL: http://lcamtuf.coredump.cx/afl/
- Oficiální projektu Perun: https://github.com/tfiedor/perun

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce:	**Rogalewicz Adam, doc. Mgr., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1. listopadu 2018
Datum odevzdání:	15. května 2019
Datum schválení:	1. listopadu 2018

Abstract

Fixing one issue sometimes brings another ten to the program. To detect these issues, especially performance issues, we often have to supply the program with input, that forces its worst-case behaviour. A popular solution to automatic inputs generation is so called fuzzing, however, its intention is to find functional bugs. In this work, we aim to construct an automatic generator of inputs whose task will be to trigger performance fluctuations. So we propose to tune fuzzing mutation rules and ways of processing the information about program run, to particularly trigger the performance bugs. We integrate our solution into a performance profile manager Perun, which stores information about every run as a profile and is able to compare these profiles to check for performance change. Therefore we can prove that executing with certain input takes more time or memory. We tested our fuzzer on several artificial projects, which shows its potential with generated inputs that prolong the runtime of the program. Such a solution would allow developers to regularly test every version of a project for performance bugs and avoid them completely by automatically finding new exhausting inputs before release.

Abstrakt

Oprava jednej chyby niekedy prináša do programu ďalších desať. Na odhalenie týchto chýb, najmä výkonnostných, často musíme programu poskytnúť vstup, ktorý vynúti jeho správanie pre najhorší prípad. Populárnym riešením pre automatické generovanie vstupov je tzv. fuzzing, avšak jeho cieľom je nájsť funkčné chyby programu. V tejto práci sa preto snažíme vytvoriť automatický generátor vstupov, ktorého úlohou bude vyvolať výkonnostné výkyvy. Navrhli sme preto vyladené fuzzing pravidlá pre mutáciu a spôsob spracovania informácií o behu programu so zámerom zachytiť výkonnostnú degradáciu. Naše riešenie je integrované do nástroja Perun, správcu výkonnostných profilov, ktorý uchováva informácie o každom behu vo forme profilu a je schopný porovnať tieto profily s cieľom detekovať zmenu vo výkone. Takýmto spôsobom môžeme dokázať, že beh programu s určitým vstupom zaberie viac času alebo pamäte. Náš fuzzer sme testovali na niekoľkých umelo vytvorených projektoch, kde ukazuje svoj potenciál generovanými vstupmi, ktoré markantne predlžujú dobu behu programu. Prínosom takéhoto riešenia je možnosť pre vývojárov pravidelne otestovať každú verziu projektu na výskyt výkonnostných chýb a vyhýbať sa im automatickým vyhľadávaním nečakaných vstupov.

Keywords

performance bugs, fuzz testing, workload mutation, worst-case, algorithmic vulnerability, denial of service

Kľúčové slová

výkonnostné chyby, fuzz testovanie, mutácia vstupu, najhorší prípad, algoritmická zraniteľnosť, denial of service

Reference

LIŠČINSKÝ, Matúš. *Fuzz Testing of Program Performance.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšírený abstrakt

Prítomnosť chýb spôsobujúcich neočakávané správanie programov je nepochybne nepríjemnou a neodvratnou súčasťou ich vývoja. Na riešenie tohto problému sa v priebehu rokov objavili rôzne typy nástrojov a metodík, ktorých hlavným cieľom bolo eliminovať (alebo aspoň znížiť) výskyt týchto chýb a poskytnúť podporu programátorom pri vývoji komplexnejších a rozsiahlejších programov.

Z hľadiska požiadavok na aspekty dnešného softvéru sa vývojári čoraz viac zameriavajú na výkonnosť programov, najmä v prípade kritických aplikácií, ako sú tie, ktoré sú nasadené v leteckom, vojenskom, zdravotníckom alebo finančnom sektore. Preto pred nasadením akéhokoľvek produktu do reálneho sveta je prirodzené a nevyhnutné zabezpečiť, aby bol dostatočne stabilný a zvládol očakávanú záťaž.

Výkonnostné chyby nie sú hlásené až tak často ako funkčné chyby a to z toho dôvodu, že zvyčajne nespôsobujú pády programov, preto je ich odhalenie náročnejšie. Navyše sa zvyknú prejavovať iba pri veľkých alebo špecifických vstupoch. Následná oprava však nebýva zložitá, a tak skutočnosť, že niekoľko riadkov kódu môže výrazne zlepšiť výkonnosť nás motivuje k tomu, aby sme venovali väčšiu pozornosť práve výkonnostným chybám už na začiatku procesu vývoja. Keďže sa počas vývoja často vydávajú nové verzie, pravidelné testovanie výkonnosti tých najnovších verzií by malo byť tým vhodným spôsobom pre včasné nájdenie problému s výkonom.

Projekt Perun je open-source nástroj, ktorý slúži na automatizovanú analýzu výkonnostných zmien na základe nazbieraného výkonnostného profilu. Okrem toho spravuje tieto profily, kde kadžý profil zodpovedá jednej verzii projektu. To pomáha používateľovi identifikovať konkrétne zmeny kódu, ktoré mohli priniesť problémy s výkonom alebo kontrolovať rôzne verzie kódu v prípade zhoršenia výkonnosti z dlhodobého hľadiska.

Neočakávané problémy s výkonom môžu viesť k vážnym zlyhaniam či k bezpečnostným problémom. Avšak manuálne testovanie výkonu nie je triviálny proces a očakáva od testerov povedomie o použitých štruktúrach a logike testovanej jednotky. Na rozdiel od toho automatizované testovanie prináša efektívnejší spôsob vytvárania testovacích prípadov, ktoré môžu spôsobiť neočakávané výkyvy výkonu v cieľovom programe. Na tento účel je vhodné prispôsobiť pokročilejšie techniky generovania testovacích dát, ako je fuzzing.

Fuzzing je testovacia technika používaná na nájdenie zraniteľností v aplikáciách ponúkaním zdeformovaných vstupných údajov a následným monitorovaním správania aplikácie. Táto agresívna technika je impozantne účinná pri hľadaní chýb a zažíva veľký úspech pri objavovaní bezpečnostných chýb. Tu vzniká myšlienka použiť fuzzing na hľadanie implementačných defektov ovplyvňujúcich výkon.

V súčasnosti existuje mnoho projektov implementujúcich techniku fuzz testovania, ale nanešťastie žiadna z nich neumožňuje pridávať vlastné stratégie mutovania, ktoré by mohli byť viac prispôsobené na cieľový program a hlavne na odhaľovanie výkonnostých slabín.

V tejto práci navrhujeme modifikáciu jednotky fuzz testovania, ktorá bude špecializovaná na generovanie vstupov chamtivých na výpočtové zdroje. Navrhujeme nové mutačné stratégie inšpirované príčinami výkonnostných chýb v reálnych projektoch a ich integrácia do Perunu predstavuje novú techniku fuzzingu. Veríme, že kombinácia výkonnostného testovania a fuzzingu by mohla zvýšiť podiel úspešne nájdených chýb počas procesu vývoja.

Naše riešenie odhalilo slabiny vo viacerých vytvorených projektoch pracujúcich s rôznymi dátovými štruktúrami a škodlivými regulárnymi výrazmi, ktorých výkon markantne degradoval pri spracovaní zmutovaných vstupov. Metodológia a výsledky tejto práce boli prezentované aj na študentskej konferencii EXCEL@FIT'19 kde boli anotované ako inovatívne so silným aplikačným potenciálom.

Fuzz Testing of Program Performance

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Doc. Mgr. Adam Rogalewicz, Ph.D. The supplementary information was provided by Ing. Tomáš Fiedor and Mgr. Bc. Hana Pluháčková. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Matúš Liščinský
May 15, 2019

Acknowledgements

I would like to thank Adam Rogalewicz for supervising this thesis, for his relaxed and constructive approach, and for willingness to help me. Furthermore, I would like to thank Tomáš Fiedor for his patience, willingness to introduce me into Perun, friendliness and readiness to answer all my questions. I must not forget to thank Hana Pluháčková for advice from statistics and Tomáš Vojnar for inspirational ideas.

Contents

Chapter 1

Introduction

'Only conducting performance testing at the conclusion of system or functional testing is like conducting a diagnostic blood test on a patient who is already dead.'

— Scott Barber

The presence of errors causing unexpected behaviour of programs is undoubtedly an unpleasant and unavoidable part of their development. To tackle this problem, various types of tools and methodologies have emerged over the years and their primary goal was to eliminate (or at least reduce) the occurrence of these defects and to provide support for programmers during development of more complex and extensive programs.

Nowadays, when talking about software aspects, developers are slowly shifting their focus more on program performance, particularly in the case of mission-critical applications such as those deployed within aerospace, military, medical or financial sectors. Naturally, before deploying anything to the real world, it is essential to make sure that it is stable enough to handle the expected load.

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. But, performance patches are usually 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. In development, new versions are frequently released, and regular performance testing of the latest releases can be a proper way of finding performance issues early.

Perun: Performance Under Control [8], is a lightweight open-source tool which includes automated performance degradation analysis based on collected performance profile. Moreover, it manages performance profiles corresponding to different versions of projects, which helps a user in identifying particular code changes that could introduce performance problems into the project's codebase or checking different code versions for subtle, long term performance degradation scenarios.

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. The reason is, that 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.

Unfortunately, manually created test cases might not detect hidden performance bugs, because it does not have to 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 just an aggressive random testing is impressively effective at finding faults and has enjoyed great success at discovering security-critical bugs as well. Using fuzz testing, developers and testers can 'hack' their systems to detect potential security threats before attackers can. So why should not we use fuzzing to discover implementation faults affecting performance?

Currently, there are many projects implementing fuzz testing technique, but 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 this work, we propose a modification of fuzz testing unit that will be specialised for producing inputs greedy for resources. We propose new mutation strategies inspired by causes of performance bugs found in real projects and integrating them within the 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. The methodology and the results of this thesis were also presented in students conference EXCEL@FIT'19 [11], and annotated as innovative with strong applicable potential.

Document structure. Chapter 2 contains a theoretical basis of fuzz testing principles, along with overview of the existing fuzz testers that inspired this work. Subsequently, Chapter 3 describes performance testing in collaboration with continuous integration and the Perun tool, which implements this principle and for which needs this work has been developed. The Chapter 4 then provides an analysis of the problem this work deals with, together with a complete draft of our solution. The breakdown of the proposed algorithm and the implementation details on which its parts are based includes Chapter 5. Finally, the Chapter 6 summarises experimental testing together with the achieved results and their analysis.

Chapter 2

Fuzz testing

Fuzzing (fuzz testing) is a form of fault injection stress testing, where a range of malformed inputs are fed to a software application while monitoring for failures [5].

The earliest reference to fuzzing dates back to 1989 when professor Barton Miller and his class developed and used a primitive fuzzer that tested the UNIX applications [22]. These roots of fuzzing are captured in the article *An Empirical Study of the Reliability of UNIX Utilities* by Miller, Fredriksen and So [14].

Fuzzing was discovered almost accidentally when one of the authors of the mentioned paper experienced electromagnetic interference when using a computer terminal during a heavy storm. This caused random characters to be inserted into the command line as the user typed, which caused a number of applications to crash. The failure of many applications to robustly handle this randomly corrupted input led professor Miller and his colleagues to develop two tools: *fuzz* and *ptyjig*[1], specifically to test application robustness to random input.

In general, *fuzz* is a random character string generator. It allows users to define the limit of generated output such as the maximum amount of characters, or by using only printable characters, etc. Tool *ptyjig* is used to supply the random input to the target utility, which input files must have the characteristics of a terminal device (e.g., the `vi` editor and the `mail` program) [14]. After each test fuzz inspects the file system looking for a core file to determine if an error has occurred. If such a file was found, it was saved together with the input that caused the error.

Whilst such a simple black-box[2] approach may sound naive, history has shown fuzzing to be surprisingly useful at uncovering faults in a wide range of software systems [13]. Nowadays, many software development teams and companies like Cisco, Microsoft, or AT&T fuzz their software on a daily basis with a purpose to find memory corruption bugs and vulnerabilities automatically [7].

Unlike, e.g. static analysis tools, it is not necessary for a fuzzer to have access to the source code of a target application in order to work. However, access to the source code may help a fuzzing framework to improve its observational capability. For example, by providing a feedback loop which drives the coverage of the different fuzzed inputs [6].

Fuzzing is a technique belonging to the group of *negative testing* (i.e. testing that the system does not do things that it is not supposed to do), as opposed to positive testing (i.e. testing that features work as specified). Software programs created for fuzz testing are

[1]Clone of fuzz and ptyjig — https://github.com/alipourm/fuzz

[2]Testing without peering into the internal structure of the component or system

commonly called *fuzzers*. Sometimes we may encounter terms used to describe tests similar to fuzzing, for example: robustness testing, protocol mutation, fault injection, syntax testing, dirty testing, or rainy-day testing [23].

2.1 The Phases of the Fuzzing

By creating *fuzz*, Miller et al. also without intent defined a model of a general fuzzer. Despite the fact that fuzzing has admittedly moved forward in the last years, all fuzzers work in the following steps:

1. **identify target**, choosing the target application which will be tested;
2. **identify inputs**, determining what inputs the target application accepts;
3. **generate fuzzed data**, basically creating new input data;
4. **execute fuzzed data**, feeding the target application with newly generated input;
5. **monitor for exceptions**, watching the target application for interesting behaviour;
6. **determine exploitability**, analysing the behaviour and classifying the input.

These phases can be performed by one unit or by various independent units and implemented using techniques with varying levels of sophistication. Each of the stages is briefly described below.

2.1.1 Identify Target

In order to maximise the effect of fuzzing it is first necessary to analyse the target software under test (SUT). The need for fuzz testing of a software depends mainly on possible risks, accessibility for an attacker or impact of the user on a system. A good example of what applications is literally essential to test are those that:

- work with valuable, personal or sensitive information;
- run in a privileged mode, higher than for common user;
- receive input over a network;
- has a specific file or library within, which are shared across multiple applications.

E.g. when a service is receiving some input from the network and is running with Windows system level privileges, it is certainly tempting for an attacker. System services and default components of operating systems represent big risk since potential successful attacks can endanger a wide range of user population [5].

Domain-specific knowledge of the target program, such as used data structures or operations provided on the input data, allows to better adjust the fuzzer for the target application, for example, by more specific and efficient methods for generating fuzz data.

2.1.2 Identify Inputs

The main reason why fuzz testing experienced such a big success is that exploitable vulnerabilities are caused mostly because the application is processing the input data vector with insufficient validation. Authors of a book *Fuzzing: Brute Force Vulnerability Discovery* [22] lists classes of inputs as follows:

- command line arguments,
- environment variables,
- web applications,
- file formats,
- network protocols,
- memory,
- COM objects,
- inter process communication.

Fuzzers can be adapted to many software areas, and help to uncover unexpected behaviour locally or even remotely. Since fuzz testing proved its quality, companies invest in the developing of specific fuzzers on a different abstraction levels and with sundry functionalities.

2.1.3 Generate Fuzzed Data

Once we identify the suitable inputs and analyse the SUT, we can generate new inputs. Fuzzers can be divided with respect to how those inputs are generated. Fuzz data can be generated using predetermined values, mutating existing data or generating data from scratch. New test cases are generated as a whole before testing, or more often iteratively generated on demand at the beginning of each test series. There exist two major categories of fuzzers: *generational* and *mutational*.

Generational Fuzzer

Sometimes called grammar-based fuzzer. Generational fuzzer generates new inputs from scratch based on a template or a grammar specification. The template defines precisely the structure of the input file that is consumed by the target program.

A template should be accurate, detailed and include all possible options for every field of the structure. This ensures that fuzzer generates valid data for control fields such as checksums or challenge-response messages and thereby achieve a high level of coverage. However, creating a bulletproof template tends to be time-consuming and complex process.

The generative method is usually used for simple models or protocols where construction of a template has no significant cost. Although many of the applications works with defined file formats or protocols (e.g. data serialisation formats, RFC standardised protocols, etc.), there is no given standard specification for templates. Hence every fuzz generator has its own design and methods for implementing the template [6].

Mutational Fuzzer

Mutational fuzzer does not require any specification of input file format. Instead it is initialised by a set of sample inputs (even one single sample file suffices). New workloads are generated by applying of mutation strategies on these initial so called *seeds*.

Mutational-based fuzzers are typically less sophisticated, however, they also require less domain knowledge such as used protocols, templates, etc. Computational work substitutes a human effort in program understanding which makes this approach cheaper.

It is worthwhile to use mutational fuzzers, e.g. when the target program uses highly structured inputs. Using mutations we do not have to accurate the entire complex structure from the beginning, but use the existing one and modify it.

A decision, whether the input (either seed or mutation) is valuable and should be reused for further work or discarded depends on several factors and it is not the functionality of fuzzer itself but of *fuzzing framework*.

Unfortunately, the SUT may reject the mutated input at the beginning of processing the data during validation since mutations can generate invalid format. Nevertheless, even invalid inputs can sometimes lead to an interesting response from SUT. Consider the example program in Listing 1, that reads a value from standard input and checks if the value is three times smaller than a magic number, and only then continues.

```c
#define MAGIC_NUM 42
int main(void) {
    int value = read();
    if (value*3 == MAGIC_NUM) {
        doWork();
    }
    else {
        err("Invalid input");
    }
}
```

Listing 1: Example of a C program that will not do any work until correct magic number is given.

If we work on an architecture where integers are stored on 4 bytes (32bits), the variable `value` can store 2^{32} different values, so the probability that it stores the correct magic number and we hence execute the body of the function `doWork()` is $1 : 2^{32}$.

Instead we could exploit the coverage information, and determine what code segments were performed during testing. Discovering new paths will cause that the input would be stored, so further mutation derived from it will have stronger potential.

Alternatively we could exploit the *symbolic execution*, a software testing technique that analyse which inputs cause each part of a program to execute. On the other hand, the main reason why symbolic execution does not outperform fuzzing is its high resource requirements, and most of the approaches do not scale on large applications.

2.1.4 Execute Fuzzed Data

After we generate new inputs we have to execute the SUT again with them. The delivery mechanism sends the generated data to the SUT input. This mechanism is closely related to a nature of the input that application is consuming, since, e.g, the system which accepts input from a file requires a different delivery mechanism than a system which accepts mouse interaction events [13]. Execution is automated and can involve opening files, sending packets, or running processes [22]. Note that the execution may take longer than previous runs with seed inputs.

2.1.5 Monitor for Exceptions

What do we acquire when we send thousands of generated requests to a server and after that, we find out that server crashed? Nothing. We do not know either which request caused the crash nor why. Hence we must monitor the program constantly, after each new test run. The monitoring system has to observe the SUT, while it processes each input into the system and tries to detect anomalies, such as errors, deadlocks or performance degradations.

The simplest method is to check the return code when the program terminates or stops for some reason. The return code may explain the cause of the system crash as each standard `signal`[3] has its own signal number associated with it. However, these signal numbers may vary on rare architectures.

More advanced methods of observation include more intrusive forms of monitoring applications, typically realised by attaching a debugger to the process [23]. One of these debugging tools is `ptrace`[4] (system call), that allows a process to inspect and control the execution of other processes. Functionality of `ptrace` relies on several tools and one of them is `strace`[5] which monitors and manipulates interactions between processes and the Linux kernel, including system calls, signal deliveries, or changes of process state. By tracking the outputs of similar tools, we can detect anomalies for different types of inputs either when opening or writing to files. For testing the memory we can list, e.g., Valgrind, Guard Malloc, Insure++, etc. Alternatively we can use `clang`[6] compiler sanitizers (ASan, TSan, MSan, and UBSan) to detect memory errors, data races, undefined behaviour, or overflows [20].

2.1.6 Determine Exploitability

The final part of fuzz testing is to analyse the potential vulnerabilities or anomalies and thorough interpretation of results. The analysis typically requires a human to determine whether the anomaly is really a vulnerability or if it is spurious. To facilitate developers work, flaws may be collected and clustered together with a report. Errors, exceptions, and their variations can be grouped into classes, which significantly reduces the total amount of vulnerabilities that the developer has to investigate.

[3]signal — http://man7.org/linux/man-pages/man7/signal.7.html
[4]ptrace — http://man7.org/linux/man-pages/man2/ptrace.2.html
[5]strace — http://man7.org/linux/man-pages/man1/strace.1.html
[6]clang — https://clang.llvm.org/

2.2 The Advantages of Fuzzing

Fuzzing is just one of plenty of techniques that can discover defects in a software, so why should one use it? Considering many existing approaches, we list several reasons why fuzzers have been widely accepted in the last years and under what terms is fuzzing the best applied.

Availability

In order to fuzz the SUT one needs only the runnable and does not need e.g. the source codes or deep knowledge of SUT architecture. Since fuzzing does not require access to the source code, it is also an open way to fuzz commercial products. The absence of source code usually exclude the use of static analysis, model checking, etc [13]. However, even though the source code is not needed for fuzzing, the analysis of the sources before actual fuzzing may yield better results, and sooner.

Simplicity

The difficulty of the fuzzing mostly depends on the character of the SUT and the level of the structure of input data. The elementary variant of fuzzing is the random data passed to the target system, which is simple to develop, especially if the program is consuming less complex inputs where the mutative approach is acceptable.

Low cost

Software engineering history has shown that test cases are more efficient when written by someone other than the original programmer since a blind spot in implementation is likely to also be repeated in testing [16]. Testers are expected to understand the implemented system and its boundary cases in order to construct effective test cases. But this approach is quite time expensive, and moreover, testers may overlook some aspects. To rely primarily only on sets of manually developed test cases is deprecated and often the test suite can be enhanced by results of performed fuzzing. Random data generation partially replaces the tester's work and will also cover cases that the tester and programmer would never consider significant [13].

Effectiveness

We finish with the most important advantage: it works. For instance, the SDL (Security Development Lifecycle) outlines fuzz testing for software verification, a mandatory policy established by Microsoft. Another example of successful fuzzing dates in April 2014, when was disclosed the Heartbleed vulnerability in the OpenSSL library, which is used by the majority of web servers. One of the most famous fuzzers AFL [25] found bugs in many tools which are listed on AFL's official website, including security-critical software such as OpenSSL; OpenSSH and nginx; Mozilla Firefox, Internet Explorer and Apple Safari; and other well-known software such as Libre Office or Adobe Flash [24].

2.3 Fuzz Testing and Performance

The fuzzing approach seems to be the right choice for testing, thus we could also use it for testing, where we won't expect an error, but that the performance of the application will deteriorate. To realise this idea, we need to modify some of the fuzz testing phases.

Generating input data should be fine-tuned to achieve better results, bearing in mind, that the problem often causes processing amount volume of a data. While monitoring the program, we are interested in the data of consumed memory, time, etc.; therefore it is necessary to select a tool able to measure them, i.e. do profiling. It will also be different to decide whether we have detected a performance issue, hence we need to compare the measured values with some baseline expected values. After comparing, we get the result, and another problem is to decide whether the comparison result shows a decrease in performance or not.

Obviously, besides fuzz testing, we can also encounter functional bugs in the tested program. Although they are not essential from the performance view, if they do not cause performance fluctuation, the output report should contain information about them as well.

2.4 Existing and Related Fuzzers

We will only list several selected fuzzers. In particular, the AFL which is the first widely used fuzzer, and PerfFuzz, which is the first attempt to tune fuzzers for finding performance bugs.

2.4.1 American Fuzzy Lop

AFL (american fuzzy lop) is an open source mutational fuzzer developed by Michał Zalewski. AFL features a colourful CLI that displays real-time statistics about the fuzzing process such as the number of found faults, hangs, average program execution speed, total AFL run time or how much time has elapsed since its most recent finds [25]. The simplified algorithm is as follows:

1. Load a queue with initial seeds.
2. Take the next input from the queue.
3. Trim the input without affecting the target's behaviour.
4. Mutate the input using selected fuzzing strategies.
5. Add the mutations deemed interesting to the queue.
6. Go to 2.

AFL can be used for both white-box[7] fuzzing (supported languages are C, C++, and Objective-C) and black-box fuzzing. Furthermore, other variations of AFL allow fuzzing projects written, e.g., in Python, Go, GCJ Java or Rust.

[7]Testing with knowledge about the internal structure of the component or system

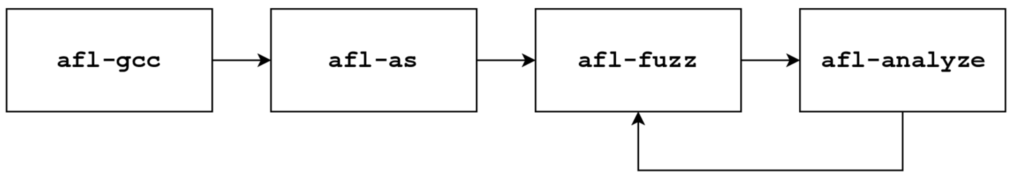

Figure 2.1: Flowchart of AFL. Redrawn from the source: [24].

Before the fuzzing, an application must be first recompiled with `afl-gcc`, a drop-in replacement for GCC or clang. Compiler output is passed to `afl-as`, a wrapper over `as`[8], that instruments the compiled target by injecting assembly code which captures branch coverage. The output of `afl-as` is an executable binary that is passed to `afl-fuzz` which fuzzes the input with the assistance of `afl-analyze`. The `afl-fuzz` element is also in charge of printing the information about current fuzzing process on user interface. The final part `afl-analyze` uses the instrumentation from `afl-as` (if provided), and observes if the execution path of the CFG was changed. It communicates only with `afl-fuzz` to improve further mutation [24][25].

AFL looks at each input file as a binary and modifies it using binary fuzzing strategies. These strategies include:

- sequential or random bit/byte flips,
- sequential or random incrementing or decrementing integer values,
- sequential or random overwriting existing data with known integers (e.g., -1, 256, 1 024, MAX_INT),
- deletion, duplication and memset of data blocks,
- splicing two distinct input files at a random location.

Successful fuzzers live and die by their fuzzing strategies. Some of AFL's strategies tend to be more successful some less, but rare feedback loop is trying to increase their efficiency, as the author of AFL, Michał Zalewski, narrates on his personal blog [26]. The feedback provided by the instrumentation injected into compiled program helps to optimise parameters of fuzzing strategies. Together with its evolutionary design of the queue provide a feedback mechanism to distinguish between insignificant mutations and those that trigger new behaviour.

This tool is precisely set to look for a variety of functional bugs and therefore can be proud of its collection of detected functional bugs. However, AFL is not adapted to looking for performance issues, neither by its mutation rules nor by program monitoring and related feedback information that affects the evolutionary design of the queue.

2.4.2 PerfFuzz

State-of-the-art mutational fuzzers are primarily focused on finding *functional bugs*. Nevertheless, recently a performance-oriented AFL variant called PerfFuzz was proposed.

PerfFuzz [10] is a coverage-guided mutational feedback-directed fuzzing engine that uses multi-dimensional feedback in the AFL's CFG graph method and additionally creates a *performance map* to improve future usability estimation of tested input. The PerfFuzz authors

[8]AS — the portable GNU assembler — http://man7.org/linux/man-pages/man1/as.1.html

defined the performance map as a function $perfmap : K \to V$, where K is a set of $\underline{\text{k}}$eys corresponding to program components (CFG edges) and V is a set of ordered $\underline{\text{v}}$alues (execution counts of CFG edges). This enables PerfFuzz to find inputs that exercise noticeable hot spots in a program and generate inputs with higher total execution path length than previous approaches by escaping local maxima. Experiments on sorting algorithm Insertion Sort, PCRE URL regular expression and others show the method is effective at generating inputs that demonstrate algorithmic complexity vulnerabilities.

Results of comparison with AFL show, that AFL initially finds a hot spot with higher execution count, but it did not grow more. On the other hand, PerfFuzz finds a hot spots with over 2× - 18× higher execution counts after 6 hours of lasting experiments [10].

Chapter 3

Performance testing

In this thesis, we aim to switch the focus of fuzzing from detecting functional bugs to uncovering performance bugs. We wish to apply the fuzzing in the performance testing — technique for determining how well has the system been designed from a performance point of view. In this chapter, we describe the basics of performance testing and its importance in the software development life cycle.

The performance testing is the process of measuring the efficiency of a software program, system or a device. This procedure usually includes quantitative tests, such as measuring the response time or the number of MIPS (millions of instructions per second) at which a system operates, and qualitative tests determining system attributes such as reliability, scalability, or interoperability. In general, performance testing verifies whether a system meets the specifications claimed by its manufacturer or vendor [17].

3.1 The Importance of Performance Testing

From the business perspective, poorly performing software programs do not commonly bring the planned benefit to an organisation, and therefore cannot be considered as a reliable asset. Regardless of causalities, this makes a bad reputation on the designers, coders, testers, and other people involved in its development process.

One can ask, when is an application considered to be performing badly or well? There does not exist any guide from generic industry standard which determines good or bad performance. Nevertheless, various informal attempts to define a standard was proposed defining e.g. *minimum page refresh time* within browser-based applications [15].

Performance testing should detect what needs to be improved before the product is released. Without it, software is probably going to suffer from problems such as poor usability, slow response when a higher number of users use it at the same time, or discrepancies across different operating systems. But, the resulting system crash can be really expensive. In August 2013, only a 5-minute downtime of Google.com was estimated to cost the search giant as much as $545 000. At the same summer, companies lost sales worth $1 100 per second because of Amazon Web Service outage [12].

3.2 Functional vs Performance Testing

Over the past decades, most of the research teams' attention has been particularly focused on the development of tools for automatically detecting *functional bugs*.

Similar tools for performance area have been developed, however, considerably less frequently, thus performance deficiencies have been perceived as less critical and often difficult to detect. But, especially in recent years, it has been shown that the severity of performance errors is in general almost comparable to functional errors, and in extreme cases, these errors can lead to practical unusability of programs, for example, when working with larger amounts of input data.

Nowadays, there are many tools that are able to detect these errors more or less satisfactorily. Usually we are talking about profiling tools, whose success depends on appropriately selected inputs. Performance bugs have the unpleasant character that their manifestation often occurs only when working with a larger volume of data, or only with specifically constructed input data. Nevertheless, if we choose too large data it can on the other hand significantly prolong the time of testing. In addition, it is often difficult to estimate the future overhead of real deployment of the product, and the errors can occur in the later phase of software life.

Conducting performance testing simultaneously with functional testing is more favourable and will add more benefits to the overall software quality. Adequate planning for conducting functional and performance testing should be done in order to keep a strong relationship among the involved parties of the project [4].

However, what if a develop company wants to sidestep the planning but still carry out the performance testing on a regular basis? Perfect solution would be to automatise this process and perform the performance testing whenever a developed project registers a significant change, e.g. when releasing a new version. In the next Chapter, we will introduce to the solutions for managing performance testing.

3.3 Perun: Performance Version System

PERUN (<u>Per</u>formance <u>Un</u>der Control) is an open-source project founded by T. Fiedor, within the VeriFIT research group. Its main objective is to automate management of program's performance profiles. It basically builds on a *version control systems*[1] *(VCS)* that are extended with the performance records (such as time or memory consumption) for each version.

Its main idea is to capture performance changes during the development by comparing performance profiles that are bound to the particular program versions. PERUN can be integrated into an already used versioning system to ensure that for each new version of the project there will be created performance profile (for example, with every commit in the versioning system).

In short, PERUN is a wrapper over existing version systems and manages profiles for different versions of particular projects. Besides that, it offers a tool suite allowing one to automate the performance regression test runs, postprocess existing profiles or interpret the results [8].

[1]systems that records changes of source code files over the time so that one can recall specific version later

3.3.1 Architecture

The implementation of Perun consists of three logically partitioned units: `data, logic and view`. The `data` part is responsible for persistent storing and managing generated performance profiles, and also for interface to supported VCS.

The importance of the `logic` part consists of management, manipulation, and automation of profile creation. It includes set of data collectors (*trace, memory, time*) for profile generation, and set of postprocessors used for profile transformation.

The `view` is a stand-alone package that provides interaction with the user using the input-output interface. This can be realised by the graphical [9] (not merged in master branch) or command line interface. At the end, the results of profiling data analysis are visualised by one of the visualisation techniques. An illustration of described architecture and selected Perun modules is in Figure 3.1.

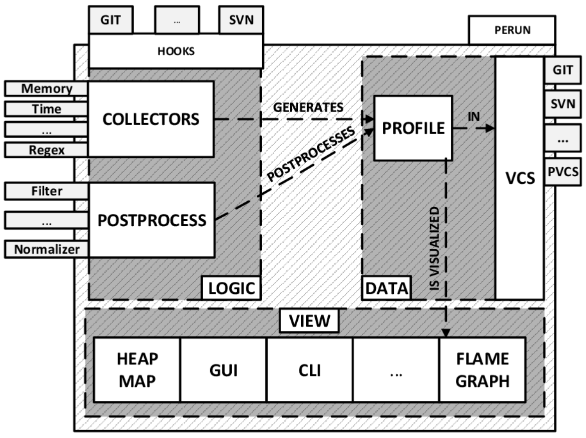

Figure 3.1: View, logic, and data together form the Perun tool architecture, adopted from [8].

3.3.2 Automatic Run of Job

Using Perun's runner infrastructure, one can run a series of steps run-collect-postprocess with defined parameters in order to generate a profile. During the profiling of application, we first collect the data by the means of profiling data collector, and further augment the collected data by ordered postprocessing steps (e.g. for filtering out unwanted data, normalising or scaling the amounts, etc.). As results we generate one profile for each application configuration and each profiling job.

Configuration of application for profiling is partitioned into three parts:

- `command`: the actual command that is being profiled, e.g. `ls`
- `arguments`: set of arguments for command, e.g. `-al`
- `workloads`: input workloads, e.g. `/usr/share`

Perun allows automatic collection of profiling data based on a pre-stored local or shared configuration, which is mainly used for regular performance analysis of project versions. The base of automation in Perun are job matrices, which are determined by a commands, arguments, workloads, collectors and postprocessors (and their internal configurations). The user can define custom matrices in the local settings (file `local.yml`), thus summarise the whole profiling process with one command `perun run matrix`. The job matrix format is shown in Listing 2. In case of irregular or specific creation of performance profiles, Perun offers the possibility to define a single job specification within options of `perun run job` command, as shown in Listing 3.

```yaml
cmds:
    - ./my_bin
args:
    - --less
workloads:
    - workload.txt
collectors:
    - name: memory
      params:
          - sampling: 1
postprocessors:
    - name: normalizer
    - name: regression_analysis
      params:
          - method: full
          - steps: 10
```

Listing 2: An example of `local.yaml` file containing a simple job matrix with required information about the selected collector, the command that will be profiled, and the other specifications such as arguments, workloads, and additional parameters for collector and postprocessor.

```
perun run job --cmd ./my_bin --args --less --workload workload.txt \
 --collector memory --collector-params memory memory-params.yaml \
 --postprocessor normalizer --postprocessor regression-analysis \
 --postprocessor-params regression-analysis ra-params.yaml
```

Listing 3: Running the job with the same configuration as in Listing 2, but using `perun run job` command. Additional parameters for collector and postprocessor are included in files `memory-params.yaml` and `ra-params.yaml`.

3.3.3 Performance Profile

Profiles store performance records collected by one of the collectors. Generated profile can be then postprocessed multiple times by any of the postprocessing units, in order to e.g. normalise or filter the values [8]. In persistent storage each generated profile is assigned to appropriate so called minor version origin (e.g. concrete commit in git VCS). A profile can be further visualised, since even a simple interpretation of outcome may be oftentimes more descriptive and lead to better understanding of program's performance. All these operations over a performance profile together symbolise the lifetime of the profile, captured in Figure 3.2.

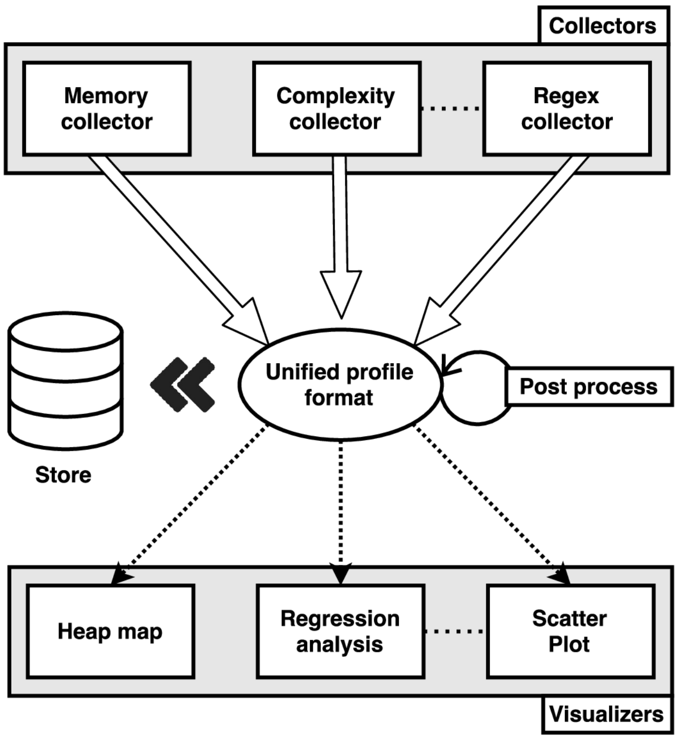

Figure 3.2: Lifetime of performance profile. Taken from [8].

Collected data are stored as a profile with format based on JSON[2]. The motivation is, that JSON-like structured data are easy to read and understandable for human and for computers which is reflected in wide support of programming languages offering an interface for operations over JSON formatted files.

Profile format requires several restrictions regarding the keys (or regions) that needs to be defined inside. Listing 4 displays the topmost structure of the profile format. Perun project documentation [8] describes format in details, we briefly outline each topmost region:

- `origin`: a hash key specifying concrete minor version of project, to which profile corresponds to. Origin links the performance records to functional changes.
- `header`: dictionary containing basic specification of the profile, like e.g. the actual command which was profiled, its parameters and input workload.
- `collector_info`: configuration of collector, which was used to capture resources and generate the profile.
- `postprocessors`: list of configurations of postprocessors in order they were applied to the profile.
- `snapshots`: list of resources that were actually collected by the specified collector.

[2]JavaScript Object Notation — https://www.json.org/

```
{
    "origin": "",
    "header": {},
    "collector_info": {},
    "postprocessors": [],
    "snapshots": [],
}
```

Listing 4: The generic scheme of profile format, adopted from [8].

3.3.4 Data Collecting and Profile Generating

Profiling data are collected by collection which generate performance profiles (i.e. the set of performance records). Perun framework currently includes these implemented collectors:

- **Trace**: based on SystemTap[3], collects running times of C/C++ functions. It is suitable to postprocess the collected data using the *regression analysis*, since they capture dependency of time consumption depending on the size of the structure. Then, we can plot individual points along with regression models using *scatter plot* visualisation technique.
- **Memory**: collects allocations of C/C++ functions, target addresses of allocations, type of allocations, etc. These collected data are suitable to visualise by the *heap map*[4].
- **Time**: a simple wrapper over the `time` utility that captures overall running time of a program.

3.3.5 Postprocessing

Once a profile is created, we can apply a sequence of postprocessing steps in order to transform its data. Perun framework currently offers five postprocessors:

- **normaliser**: normalises the resources of the same type to the interval (0, 1), where 1 corresponds to the maximal value of the given type.
- **regression analysis**: attempts to find the fitting model (linear, quadratic, logarithmic, etc.) for a dependent variable based on another independent one. E.g. the dependency of function runtime depending on the size of the underlying structure.
- **clusteriser**: tries to classify resources to uniquely identified clusters or to group similar amounts of resources.
- **regressogram**: non-parametric method, which tries to fit models through data by dividing the interval into N equal buckets, where a bucket value is a result of selected statistical aggregation function (mean/median).

[3]SystemTap — https://sourceware.org/systemtap/documentation.html
[4]graphical representation of data which values contained in a matrix are represented by colours

- **moving average**: non-parametric approach, which uses the analysis of data points by creating a series of values based on the specific aggregation function; values are derived from different subsets of the full data set.
- **kernel regression**: non-parametric technique that estimates the conditional expectation of a random variable by placing a weighting function (kernel) over each estimated data point.

3.3.6 Automatic Detection of Performance Changes

Perun offers an automatic check for performance changes between two isolate profiles (so called *baseline* and *target* profile), with the same configuration (i.e. collected by same collectors, postprocessed using same postprocessors, and collected for the same combination of command, arguments and workload). These profiles may be registered in index (i.e. assigned to the concrete minor version), stored in pending profiles or simply stored in the filesystem. Usually the baseline corresponds to previous stable version (e.g. the previous head) and target to new untested version (e.g. new head or commit).

For such a pair of target and baseline profiles, we can use several methods, which can then report multiple performance changes. Potential changes of performance are then reported for these pairs of profiles, together with more precise information. This information then helps a developer to evaluate whether the detected changes are real or spurious.

Perun framework currently supports the following strategies for detection of the performance changes:

- **Average Amount Threshold**: computes averages for each unique group of resources, and consider them as a representation of the performance. Each average of the target is then compared with the average of the baseline and if their ratio exceeds a certain threshold interval, the method reports the change (optimisation or degradation).
- **Best Model Order Equality**: checks for each unique group of resources, whether the best performance (or prediction) model has changed. The result can be e.g. that the best model changed from linear to quadratic.
- **Fast Check**: simple method, based on the subtraction of best-fit models and subsequently interleaving of these data by newer models.
- **Linear Regression**: heuristic based on the results of linear regression models, which models the relationship between independent variables x and dependent variables y as function $y = b_0 + b_1 \cdot x$. The heuristic compares the coefficients b_0 (*y-intercept*) and b_1 (*slope*).
- **Polynomial Regression**: represents the change in a form of n^{th} degree polynomial function [21]. This heuristic tries to find the best fit n^{th} degree polynomial for subtraction of best baseline and target models.

To summarise, Perun allows automatic detecting of performance changes between various minor versions within the history with the aim to protect the project from potential performance degradation.

Chapter 4

Analysis and Design

The underdeveloped field of performance fuzz testing has inspired us to explore this issue more and extend the Perun tool with fuzzing module that will try to find new workloads (or inputs) that will likely cause a change in program performance. We will start with a motivational example as an introduction to the problem, then we analyse the problem and finally we will propose the solution together with a short explanation of the principles.

Often, the overall performance of a program highly depends on its input data (if it consumes any). Although manually written tests can cover even 100% of the code, test cases may not reveal hidden vulnerabilities until the unusual input data are provided.

```c
#include <stdio.h>
#include <stdlib.h>
#define DIGITS 2

void doSomething(void){ return; }

int main(void){

    FILE * fp = fopen("workload.txt","r");
    char array[DIGITS];
    for(int i=0; i<DIGITS; i++)
        array[i] = fgetc(fp);

    unsigned number = atoi(array);
    for(unsigned i=0; i<number; i++)
        doSomething();
}
```

Listing 5: Example C program that shows a vulnerability when signed integer is assigned to unsigned integer variable.

In Listing 5 one can see an example program, which reads two characters from an input file (expecting it contains numerical values), stores them in an array and then converts the array to an integer using standard `atoi`[1] (*array to integer*) function. The original in-

[1] atoi — https://en.cppreference.com/w/cpp/string/byte/atoi

tention was to avoid large numbers and only take two digits into account, so the number should be out of interval $< 0, 99 >$. But, this solution contains hidden vulnerability. On the highlighted line, the result of converting is assigned to **unsigned** integer variable, but the return value of `atoi` function is a **signed** integer. In case that the input file will contain for example string '-1', `atoi` will successfully convert the string to an integer -1, which is represented as `0xFFFF FFFF` in hexadecimal (on architecture where integers are stored on 4 bytes). Considering that the variable `number` is defined as unsigned integer, the following loop will call `doSomething` function `UINT_MAX`(2^{32}-1) times leading to performance degradation.

4.1 Problem Analysis

Basically, the goal of this work is to generate new input data that could possibly exercise (i.e. consume as many resources or time as possible) the target program the most. We believe that employing the fuzz technique could help create such new input data. We propose that for the purpose of lightweight fuzz testing mutational methodology is more preferable. Mutational strategies should be more oriented and tuned for performance. Although traditional mutation strategies were built rather for finding functional faults, certainly it is good to combine them with the performance tuned ones.

In conjunction with Perun tool, the approach of regular performance testing, the user could find with each new version new workloads that cause a problem and keep track of the progress of project performance power over time. After fixing the bug of certain performance issue revealed by the fuzzer, the user is able to test the target application performance again either with the worst-case workloads assigned to earlier versions or by repeatedly performing the fuzz testing. Because fixing one bug may sometimes create new ones.

4.2 Requirements for Fuzz Unit

In this section we briefly summarise the functional requirements and specifications of the resulting product.

1. New mutation rules. The product must offer new, reasonably designed and performance affectable rules. The group of rules need to be general, not focusing on the only one type of potential performance problem.

2. Classic rules. The existing fuzzers have implemented them, and they have achieved the success, therefore it is advisable to add some classic generally used mutation rules to our collection of rules.

3. Perun influence. This means selecting inputs for mutation mainly according to the Perun results, because it is the main difference from the existing performance fuzzers.

4. Workload picking based on coverage. Since the fuzzing is a brute-force technique, we do not want to test with Perun every workload, just interested in terms of amount of executed code. Note that Perun testing would be often unnecessary and process of collecting, postprocessing and detection brings a considerable overhead.

5. Interpretation of workloads. We think that after finishing the fuzz testing, testers primarily want to know what workloads are making the troubles to application and how they differ from the original files.

6. Interpretation of fuzzing. For better imagination of the finished fuzzing process, fuzzer should offer visualised information about it that can be helpful for future fuzzing.

4.3 Design of Performance Fuzzer

We have already described the general fuzz testing in Section 2.1. The described steps must be implemented accordingly to what the unit should be focused on. In this work we construct a lightweight *Mutation Based Fuzzing Tool* tuned for detecting performance changes, i.e. performance optimisations and degradations.

The proposed solution will be modifying *files* (one of the most common format of program workload). We believe that the *mutational* approach is more suitable in order to create new workloads. Existing projects inspired us to implement *the feedback loop* with coverage information, for the purpose of increasing the efficiency and chances to find the worst-case workloads. Another feedback will be obtained from PERUN, which automatically detects performance changes based on the data collected within the program runtime.

4.3.1 General Description of the Algorithm

In this section we will describe design of performance fuzz tester. Its main loop is depicted in Listing 6.

An inevitable element for starting the fuzzing is to collect suitable set of sample *seed* inputs (or workloads), also called *input corpus*. In classical fuzzing methods we work with so called inputs, however, in this work we will adapt the terminology of PERUN, which calls the input of programs the *workloads*. The *seeds* should be valid workloads for the target application, so the application terminates on them and yields expected performance. Collecting workloads into the corpus is done by pseudo function `get_initial_corpus` within the overall performance fuzzing algorithm captured in Listing 6. In our fuzzer, the seeds will be provided by the user.

For different file types (or those of similar characteristics) we want to use different groups of mutation methods (function `choose_rules_according_to_filetype`) as described in Section 4.4. The 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 errors more quickly.

Before running the target application with newly generated malformed workloads, it is necessary to first determine the *performance baseline*, i.e. the expected performance of the program, to which future results (so called targets) will be compared. In initial testing we first measure code coverage (number of executed lines of code) while executing each initial seed. The median of measured coverage data is then considered as the baseline for coverage testing (`base_cov` variable). Second, PERUN is run to collected memory, time or trace resource records with initial seeds resulting into baseline profiles (`base_profile`). Practically *performance baseline* is a profile describing the performance of the program on the given workload corpus. After the initial testing, the seeds in the corpus are considered as parents for future mutations and rated by the evaluation function.

Once we assemble initial seeds, we can start the actual fuzzing. The fuzzing loop itself starts with choosing one individual file from corpus (function `choose_parent`) using heuristic described in Section 5.6.1. This file is then transformed into mutations (function `fuzz`) and their quantity is calculated using dynamically collected fuzz stats (see Section 5.6.2 for more details). We test every mutation file with the goal to achieve maximum possible code coverage. We first focus on gathering the interesting workloads, which increase the number of executed lines. We argue that coverage based testing is fast and can yield satisfying results. Later we will combine these results with the performance check, which is slower. In case that, code coverage exceeds the certain threshold, responsible mutation file joins the corpus and therefore can be fuzzed in future to intentionally trigger more serious performance issue. Each parent joining the corpus gets rated, in this phase only according to reached coverage.

```
results = []
corpus = get_initial_corpus()
mutation_rules = choose_rules_according_to_filetype(corpus)
base_cov = init_cov_test(corpus)
base_profile = init_performance_test(corpus)
rate_parents(corpus)
# Fuzzing loop
while timeout not reached:
   interesting workloads = []
   # Coverage-guided testing
   while execs_limit not reached and collected_files_limit not reached:
      candidate = choose_parent(corpus)
      muts = fuzz(candidate, mutation_rules, fuzz_stats)
      # Gathering interesting mutations
      interesting_workloads += test_for_cov(muts, base_cov, icovr_ratio)
      corpus += interesting_workloads
      rate_parents(interesting_workloads)
      update_stats(fuzz_stats, interesting_workloads)
   adapt_icovr_ratio(icovr_ratio)
   # Profile-guided testing
   results += test_with_perun(interesting_workloads, base_profile)
   update_rates(results)
   update_stats(fuzz_stats, results)
```

Listing 6: Pseudocode of Performance Fuzzing Algorithm.

After gathering the interesting workloads, the fuzzer collects run-time data (memory, trace, time), transforms the data to a so called target profile and checks for performance changes by comparing newly generated target profile with baseline performance profile (see [18] for more details about degradation checks). Then the tested workloads rates have to be recomputed to include the performance change result (function `update_rates`). The intuition is, that running coverage testing is faster than collecting performance data (since it introduces certain overhead) and collecting performance data only for possibly newly covered paths could result into more interesting workloads. According to the number of gathered workloads we adapt the coverage increase ratio, with an aim to either mitigate or tighten the condition for classification a workload as an interesting one.

List of results of each testing iteration in the main loop contains successful mutations and the history of the used rules, that led to their current form. This information is updated after each test run to make the best decisions at any time. Moreover, collecting interesting workloads is limited by two variables: the current number of program executions (`execs_limit`) and the current number of collected files (`collected_files_limit`). The first limit guarantees that the loop will terminate. On the other hand, this limit of executions could be set to excessively high value, which would lead to a long duration of this phase, especially if the test program itself is used to run for a longer time. The second limit ensures the loop will end in reasonable time and collects reasonable number of workloads.

4.3.2 Absence of Source Files

We can collect line coverage only in the presence of source files. Nevertheless, the fuzzer should provide fuzz testing even without them. In that case we skip the first (and fast) testing phase and only checks for possible performance changes. In Listing 7 is captured an algorithm in pseudocode, relying only on results of PERUN's detection of performance change.

```
results = []
corpus = get_initial_corpus()
mutation_rules = choose_rules_according_to_filetype(corpus)
base_profile = init_performance_test(corpus)
rate_parents(corpus)
# Fuzzing loop
while timeout not reached:
   candidate = choose_parent(corpus)
   muts = fuzz(candidate, mutation_rules, fuzz_stats)
   # Profile-guided testing
   results += test_with_perun(muts, base_profile)
   corpus += results
   rate_parents(results)
   update_stats(fuzz_stats, results)
```

Listing 7: Variation of pseudocode of Performance Fuzzing Algorithm without the access to source files.

4.4 Mutation Strategies

In general, the goal of mutational strategies is to randomly modify a workload to create a new one. We will present a series of rules inspired by performance bugs found in real projects, and general knowledge about used data structures, sorting algorithms, or regular expressions.

Both the types of workloads and the rules for their modification are divided into two basic groups: *text* and *binary*. In addition, we added specific rules for XML format based files. Each rule has its own label name (T stands for text, B for binary and D for domain-specific), with a brief description of what it concentrates on and the demonstration result of its application on some sample data.

4.4.1 Text File Strategies

The following rules are constructed strictly for text files. Suppose the seed workload for fuzzing is the file with the string:

`‘the quick brown fox jumps over the lazy dog’`.

Rule T.1: Double the size of a line. This rule focuses on possible performance issues associated with long lines appearing in files. The inspiration comes from the gedit[2] text editor, which shows signs of performance issues when working with too long lines even in small text files. Another potential performance issue that this rule could force is a poorly validated regular expression that could be forced into lengthy backtracking while trying to match the whole line.

```
‘the quick brown fox jumps over the lazy dogthe quick brown fox jumps over
the lazy dog’
```

Rule T.2: Duplicate a line. Similar to the previous rule, but instead extends the file vertically. Suppose that there is a line in a file that represents a performance vulnerability but does not manifest in small sizes, therefore the degradation would not be detected. By multiplying the line we can likely trigger the vulnerability and this could lead to a decline in performance.

```
‘the quick brown fox jumps over the lazy dog
the quick brown fox jumps over the lazy dog’
```

Rule T.3: Divide a line. Similarly to Rule T.2, the rule may pose a threat to programs, whose performance does not depends so much on the length of the line as the number of lines in the workload file. Moreover, the rule can be effective for regular expressions matching whole lines. The line will be cut, which means it will not contain what the regular expression would expect, and could force backtracking.

```
‘the quick brown fox jumps o
ver the lazy dog’
```

[2]gedit — https://wiki.gnome.org/Apps/Gedit

Rule T.4: Change random character. This is traditional fuzzing method since the emergence of fuzzing, which can trigger unexpected behaviour for various reasons. While this is not a specific rule for performance, in PerfFuzz [10] authors found interesting workloads even with basic mutation rules.

`‘the qu1ck brown fox jumps over the lazy dog’`

Rule T.5: Repeat random word of a line. On vulnerabilities, e.g., in a handler when a program is trying to store what has been read and the record already exists (hash table, or user registration to a database). Further, in situations where the program expects unique input data, e.g. sorting algorithm QuickSort reaches its worst-case when all the elements are the same [3].

`‘the quick brown fox jumps over the lazy dog dog dog dog dog dog dog dog’`

This pair belongs to the rules that focus mainly on sorting algorithms and searches in data structures. According to [3], QuickSort exhibits worst-case $O(n^2)$ behaviour also when the elements are sorted or reversely sorted. We expect that similar behaviour could hold for the other sorting algorithms, searching algorithms (and their heuristics), and others which assume randomly sorted workload. The result is showing which words change their position within the line and which not when sorting in ascended and descended order.
Rule T.6: Sort words or numbers of a line.

`‘brown dog fox jumps lazy over quick the the’`

Rule T.7: (Reversely) sort words or numbers of a line.

`‘the the quick over lazy jumps fox dog brown’`

The following rules are focused on the efficiency of the program white character handling. The inspiration lies in the well-known StackOverflow outage on July 20, 2016. The reason of the outage was regular expression `^[\s\u200c]+|[\s\u200c]+$` intended to trim unicode space from start and end of a line. If the string to be matched against contains e.g. 20 000 space characters in a row, but after the last one there is a different character, Regex engine expected a space or the end of the string. Realising it cannot match like this it backtracks, and tries matching starting from the second space, checking 19,999 characters, then from third space and so on [2]. Similar deployment of the seemingly harmless regular expression could be detected with the help of these rules.

Rule T.8: Append whitespaces. Sometimes we want to trim a line, i.e. remove the white characters from the front or back. This rule simply adds 100 to 1 000 whitespaces at the end of the line. The amount of whitespaces is chosen from the same interval for every rule in this group.

`‘the quick brown fox jumps over the lazy dog␣␣␣␣␣␣␣␣␣␣␣␣...␣␣␣␣␣␣␣␣␣␣␣␣’`

Rule T.9: Prepend whitespaces. A follow-up rule that adds white characters to the beginning of a line.

`‘␣␣␣␣␣␣␣␣␣␣␣␣...␣␣␣␣␣␣␣␣␣␣␣␣the quick brown fox jumps over the lazy dog’`

Rule T.10: Insert whitespaces on a random place. This mutation can split data into multiple parts. For applications relying on CPU caching this rule could force load of gaps in the memory (which are often useless data), and therefore application may slow down.

‘the quick brown fox jum␣␣␣␣␣␣␣␣␣␣␣␣···␣␣␣␣␣␣␣␣␣␣␣␣ps over the lazy dog’

Rule T.11: Repeat whitespaces. Follows the same principle as the previous rule, with the difference that spaces will be in the same place only larger. If the input has a more strict format, then the previous rule will not succeed because it breaks the input data format. In this case the spaces will be multiplied and the structure may not necessarily be corrupted.

‘the quick brown fox jumps over the␣␣␣␣␣␣␣␣␣␣␣␣···␣␣␣␣␣␣␣␣␣␣␣␣lazy dog’

Rule T.12: Remove whitespaces of a line. This method removes any white spacing of a line, and thereby creates continuous data. When using a hash table, two complications can occur: (a) the hash function could calculate the index for a long time, (b) always new unique data could quickly fill the table, and thereby enlarging the hash table. A similar case is when a program expects a space after e.g. 10 characters and it is missing in the file.

‘thequickbrownfoxjumpsoverthelazydog’

The traditional rules that deletes random parts of the data are inspired by fuzz testing cores. Removing of some elements may lead to, e.g., the parser waiting for some character or string or number. This rule could also be effective in the case of regular expression backtracking, again.

Rule T.13: Remove random line.

‘~~the quick brown fox jumps over the lazy dog~~’

Rule T.14: Remove random word.

‘the quick brown fox jumps over the ~~lazy~~ dog’

Rule T.15: Remove random character.

‘the quick brown f~~o~~x jumps over the lazy dog’

4.4.2 Binary File Strategies

We propose the following rules for binary files. In case of binary files we cannot apply specific domain knowledge nor can we be inspired by existing performance issues instead we mostly adapt the classical fuzzing rules. Let us assume binary file with the following content:

‘This is !binary! file.\0’.

The following two rules are based on the fact, that in C language, the string is considered to be a series of characters terminated with a NULL character '\0'. Thus, a string cannot contain a NULL character and by adding it and then reading can terminate the program thinking it reached the end of a string and the read data will be incomplete. Removing the zero byte could lead to program non-termination or crash reading the whole memory.

Rule B.1: Remove random zero byte.

`'This is !binary! file.`~~`\0`~~`'`

Rule B.2: Add zero byte to random position.

`'This is !\0binary! file.\0'`

The inspiration for the last binary fuzzing rules is their deployment and success in existing fuzzing tools. Although they do not have a specific focus on performance, they can often trigger unexpected behaviour.

Rule B.3: Insert random byte.

`'This is !binar$y! file.\0'`

Rule B.4: Remove random byte.

`'This is !`~~`b`~~`inary! file.\0'`

Rule B.5: Byte swap.

`'This is ebinary! fil!.\0'`

Rule B.6: Bite flip.

`'This is %binary! file.\0'`

4.4.3 Domain-Specific Strategies

If we have more domain-specific knowledge about the workload format we can devise specific rules. For the purpose of finding potential vulnerability more quickly, we want to avoid workload discarding at the potential initial check. We propose rules for removing tags, attributes, names or values of attributes used in XML based files (i.e. `.xml`, `.svg`, `.xhtml`, `.xul`). For example, we can assume a situation, when fuzzer removes closing tag, which will increase the nesting. Then a recursively implemented parser will fail to find one or more of closing brackets (representing recursion stop condition) and may hit a stack overflow error. Let us assume a sample line of XML file:

< book id='bk106' pages='457' >

Rule D.1: Remove an attribute.

< book id='bk106' ~~pages='457'~~ >

Rule D.2: Remove only attribute name.

< book id='bk106' ~~pages=~~'457' >

Rule D.3: Remove only attribute value.

< book id='bk106' pages='~~457~~' >

Rule D.4: Remove a tag.

~~< book id='bk106' pages='457' >~~

We can adapt similar rules for e.g. HTML files or JSON-format. In this work we limit ourselves to XML only. The concept of how the individual rules are selected, when the rule is preferred and the other is neglected (or totally rejected) is described in Section 5.3.

Fuzzer also offers the possibility of adding custom rules. For adding the rules to a mutation strategy set, one has to launch the fuzzer with a special file in YAML file format containing the description of these rules. YAML is chosen because the Perun tool already includes ancillary functions for basic work with YAML files. Each rule is represented as an associative array in a form *key: value*, where both are regular expressions but *key* is a pattern which should be replaced, and *value* is the replacement. An example of how such a file might look like is shown in Listing 8.

```
Back: Front
del: add
remove: create
([0-9]{6}),([0-9]{2}): \\1.\\2
(\\w+)=(\\w+): \\2=\\1
```

Listing 8: A file containing five custom rules defined by regular expressions. Each rule is then implemented in a separate function, where occurrences are substituted by standard regular expression replacing.

Chapter 5

Implementation

In previous chapter we proposed a fuzzer with focus on triggering performance bugs. This work will be integrated in the PERUN in Python 3.5. In this chapter, we will describe the options of fuzz unit incorporating in the PERUN, and reveal selected implementation details of the Performance Fuzzing Algorithm from Listing 6 and 7, respectively, and some other heuristics and features.

5.1 Fuzzer Implementation Structure

The proposed solution required to split the implementation part into several logical units. We have broken the functionality of the fuzzer into the following nine modules:

- `coverage.py`: implements functions for coverage-guided testing,
- `factory.py`: main module of the project, contains the fuzzing loop, controls mutating, rating the parents and so on,
- `filesystem.py`: contains functions dedicated for various operations over files and directories in file system which are helpful for fuzzing process,
- `filetype.py`: module for automatic recognising the file type and choosing appropriate fuzzing rules, and handling with user defined rules,
- `interpret.py`: contains a set of functions for interpretation the results of fuzzing,
- methods/`binary.py`: collects general fuzzing rules for binary files,
- methods/`textfile.py`: collects general fuzzing rules for text files,
- methods/`xml.py`: collects fuzzing rules specific for XML files.

If a user wants his custom rules to become a part of the default set of rules (for certain type of file), it is necessary to implement them and modify the script `filetype.py`, which is responsible for selecting the rules. To add, for example, specific rules for JSON file type, one just has to create a new script, say `json.py`, and modify the rules selection. Note that every rule should contain a brief description, which will be displayed after fuzzing.

Integration within Perun. The task of proposed fuzzer, as part of the Perun tool, is to find the potential harmful workloads during continuous performance testing. Integration of fuzz unit within Perun is captured in Figure 5.1.

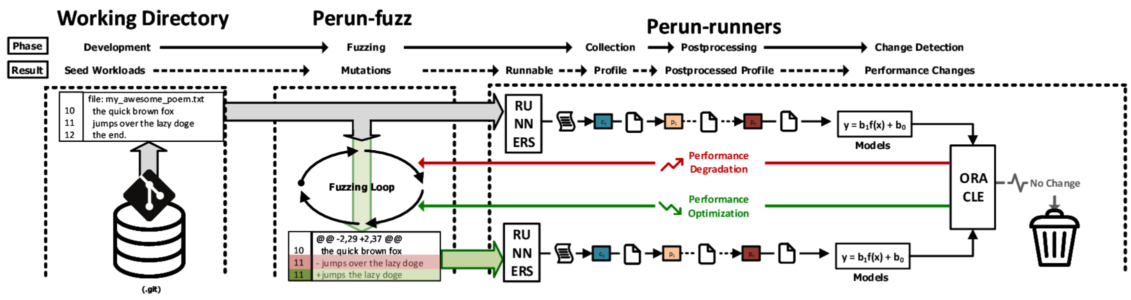

Figure 5.1: Fuzzer incorporated into the Perun, adopted from [8]. Fuzzer unit takes the seed workloads (e.g. problematic workloads for previous version of project) and starts its loop. In order to evaluate new mutations, uses the results of analyses yielded from Perun performance testing. If the fuzzer generates the workload which triggers a performance degradation, and so Perun detects it, workload is stored, therefore developers can fix this performance issue and keep the workload for future testing.

5.2 Acquiring Initial Seeds

We first have to get the set of user-provided initial sample workloads (i.e. workload corpus): a crucial aspect of mutational fuzzing. Workloads can be passed to fuzzer comfortably as an arbitrary mix of files or directories. Directories are then iteratively walked for all files with reading permissions and optionally name matching user specified regular expression. For example, consider an application that works with text files (in format of TXT, XML, HTML) and user has one large directory with various collection of workloads. We can fuzz with XML files just with simple regular expression `^.*.xml$`. If we want to skip all the files with the name containing string „error“ we can use `^((?!error).)*$`. Note that the fuzzer should always be launched with just one type of initial files even if the target application supports more types, since we tune the rules according to workload file format.

5.3 Mutation Methods Selection

The resulting fuzzer distinguishes between text and binary files and for each format defines a set of concrete mutation strategies. It can be further extended by other strategies based on file mime-type as well. We select corresponding strategies on the beginning, based on the first loaded workload file. Basically, if this file is a binary, all the rules specific to binaries are added to the set of rules, otherwise we add all the basic text rules. If the mime type of a file is supported by the fuzzer, we add to the set of rules mime-specific rules as well as any user-defined rules. Note that the group of currently supported specific methods for certain types can be further expanded by other file types.

We argue the advantage of fuzzing with one file type rests in its code covering feature. To be more precise, we are not observing at the overall percentage of code coverage, but how many lines of code has been executed in total during the run, with an aim to maximise it. Consider an application that extracts meta-data from different media files, such as WAV, JPEG, PNG, etc. If a PNG image file is used as a seed to this application, only the parts related to PNG files will be tested. Then testing with WAV will cause, that completely different parts of the program will be executed [6], hence total executed code lines of these two runs cannot compare with each other because reaching higher line coverage with WAV files would lead to preferring them for fuzzing, and PNG files would be neglected (see Section 5.6.1 for more information about file preference). Moreover, we are aware that this strategy may miss some performance bugs. Fuzzing multiple mime-types is current feature work.

5.4 Initial Program Testing

Baseline results (i.e. results and measurements of workload corpus) are essential for detecting performance changes because newly mutated results have to be compared against some expected behaviour, performance or value. Hence, initial seeds become test cases and they are used to collect performance baselines. By default, our initial program testing as well as testing within the fuzzing loop (Section 5.6) interleaves two phases described in more details below: coverage and performance-guided testing.

5.4.1 Coverage-Guided Testing

If one wants to achieve good results in triggering performance changes it is generally recommended to monitor the code coverage during the testing especially tracking coverage of unique paths. The intuition is that by monitoring how many paths are covered and how often they are executed, we can more likely encounter a new performance bug.

In our fuzzer, we use Gcov tool to measure the coverage. The program has to be build for coverage analysis with GNU Compiler Collection (GCC) with the option `--coverage` (or alternatively a pair of options `-fprofile-arcs -ftest-coverage`). The resulting file with the extension `.gcno` contains the information about basic block graphs and assigns source line numbers to blocks. If we execute the target application a separate `.gcda` files are created for each object file in the project. These files contain arc transition counts, value profile counts, and additional summary information [1].

Gcov uses these files for actual profiling which results into the output `.gcov` file. Version 4.9 supports easy-to-parse intermediate text format using the option `-i` when launching the tool. However, older versions does not support this option, hence before the run, we have to dynamically check the version and accordingly parse the output files. The difference between intermediate and standard format of output file is shown in Listings 9 and 10.

Total count of executed code lines through all source files represents the coverage (and partly also a performance) indicator for the first testing 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 was likely 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. Support of more precise coverage metrics is a future work.

So first the target program is executed with all files from workload corpus. After each single execution, `.gcda` files are filled with coverage information, which Gcov tool parses and generates output files. We parse coverage data from the output `.gcov` file, sum up line executions, compare with the current maximum, update the maximum if new coverage is greater and iterate again. It follows that base coverage is the maximum count of executed lines reached during testing with seeds.

```
file:motivation-example.c
function:6,10,doSomething
function:8,1,main
lcount:6,10
lcount:8,1
lcount:10,1
lcount:12,3
lcount:13,2
lcount:15,1
lcount:16,11
lcount:17,10
```

Listing 9: The resulting gcov file in intermediate text format with information about run of motivation example from Listing 5. We parse the lines starting with `lcount`, where the first value means the number of the line and the second how many times was the line executed. The program was launched with an input string '10', which led to 39 executed lines in total.

```
    -:    0:Source:motivation-example.c
    -:    0:Graph:motivation-example.gcno
    -:    0:Data:motivation-example.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:#include <stdlib.h>
    -:    3:
    -:    4:#define DIGITS 2
    -:    5:
   10:    6:void doSomething(){ return; }
    -:    7:
    1:    8:int main(int argc, char ** argv){
    -:    9:
    1:   10:    FILE * fp = fopen("workload.txt","r");
    -:   11:    char array [DIGITS];
    3:   12:    for(int i=0; i<DIGITS; i++)
    2:   13:        array[i] = fgetc(fp);
    -:   14:
    1:   15:    unsigned number = atoi(array);
   11:   16:    for(unsigned i=0; i<number; i++)
   10:   17:        doSomething();
    -:   18:}
```

Listing 10: The resulting gcov file in standard format with information about run of the same program with the same input as in the previous case. One can see, that parser will have to process twice as many lines in comparison with the intermediate format, because of additional information and code, which are currently unnecessary for our analysis. Therefore the parser has to go through 23 lines but only 8 of them contained wanted information.

5.4.2 Profile-Guided Testing

While coverage-based testing within fuzzing can give us fast feedback, it does not serve as an accurate performance indicator. We hence want to exploit results from Perun. 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 profile (i.e. the set of performance records). Analogically to the previous section, we will need a performance baseline, which will be compared with newly generated mutations. Profiles measured on fuzzed workloads (so called *target profiles*) are then compared with a profile describing the performance of the program on the initial corpus (so called*baseline profiles*). 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 the least-squares method. We then use the Perun internal degradation methods [18] which work as follows. From both of these sets we select for each function models with the highest value of *coefficient of determination* R^2. This coefficient represents how well the model fits the data, and also 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 the 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 classifiers, which returns detected degradations for each function.

5.5 Parents Rating

Initially, the workload corpus is filled with seeds (given by user), which will be parents to newly generated mutations (we can also call these seeds *parent workloads*). While we fuzz, we extend the corpus with successful mutations which become *parent workloads* too. The success of every workload is represented by the *fitness score*: a numeric value indicating workload's point rating. The better rating of workload leads either to better code coverage (and possibly new explored paths or iterations) or to newly found performance changes. We calculate the total score by the following evaluation function:

$$score_{workload} = icovr_{\text{workload}} * (1 + pcr_{\text{workload}}).$$

Increase coverage rate (icovr): This value indicates how much coverage changed if we run the program with the workload, compared to the base coverage measured for initial corpus. Basically, it is a ratio between coverage measured with the mutated workload and the base coverage:

$$icovr_{\text{workload}} = cov_{\text{workload}}/cov_{\text{base}}.$$

Performance change rate (pcr): In general, we compare the newly created profile with the baseline profile (for details see Section 5.4.2) and the result is a list of located performance changes (namely *degradations*, *optimisations* and *no changes*). Performance change rate is then computed as ratio number of degradations in the result list:

$$pcr_{\text{workload}} = \text{cnt}(degradation, result)/\text{len}(result)$$

This value plays a large role in the overall ranking of workload, because it is based on the real data collected from the run. And so workloads that report performance degradations and not just increases coverage have better ranking. The computation of pcr_{workload} could further be extended by the rate of degradations, i.e. if two workloads found the same number of degradations, the workload which contains more serious change would be ranked better. Optimisations of ranking algorithm is another future work. This evaluation serves for informed candidate selection for fuzzing from the parents, described in the Section 5.6.1.

5.6 Fuzzing Loop

This main loop runs for a limited time specified the user. One, however must take take into account that testing and especially performance analysis has some overhead and so it may sometimes take longer. On the other hand, the program can catch SIGINT signal to terminate the fuzz test when a user decides to quit earlier. Fuzz unit is ready to receive this signal, however, other Perun units (collectors, postprocessors) have not implemented handlers for interruption signal, hence it is not recommended to interrupt during performance testing, but only in the coverage-guided testing phase. In this section, we described the main loop of the whole fuzzing process and some of its most significant parts.

5.6.1 Parent Workload Selection

The first task at the beginning of every iteration is to select the workloads from parents which will be further mutated. All parents are kept sorted by their scores, and the selection for mutation consists of dividing the seeds into five intervals such that the seeds with similar value are grouped together. Five intervals seem to be appropriate because with fewer intervals parents are in too big groups and in case of more intervals, parents with similar score are pointlessly scattered. First, we assign a weight to each interval using linear distribution. Then 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. The process of selecting is illustrated in Figure 5.2. The intuition behind this strategy is to select the workload for mutation from the best rated parents. From our experience, selecting only the best rated parent in every iteration does not led to better results, and other parents are ignored. Hence we do selection from all the parents, but the parent with better score has a greater chance to be selected.

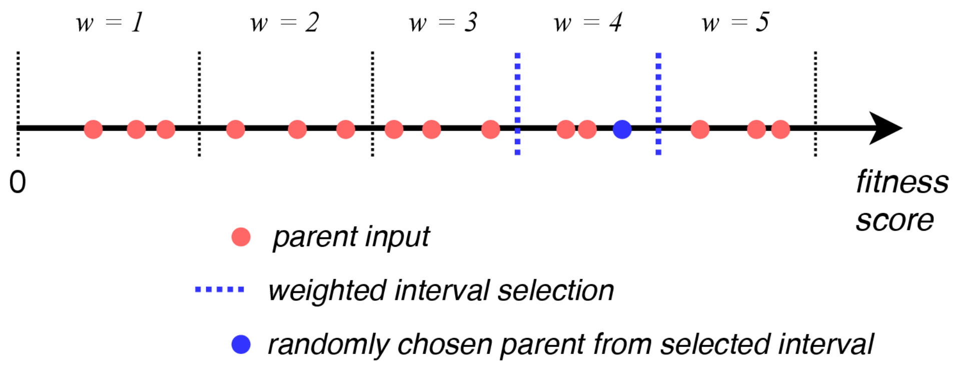
w = 1
w = 2
w = 3
w = 4
w = 5
0
fitness score
parent input
weighted interval selection
randomly chosen parent from selected interval

Figure 5.2: 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.

5.6.2 Data Mutating

Once we have baseline data for workload corpus and choose appropriate mutation rules for concrete file type, we use fuzzer to gradually apply the mutations and generate new workloads. However, it is necessary to determine how many new files (N) to generate by rule f in the current iteration of fuzzing loop. If N is too big and we generate mutations for each rule f from the set of rules, the corpus will bloat. On the other hand, if N is too low, we might not trigger any change. Instead we propose to dynamically calculate the value of N according to the statistics of fuzzing rules during the process. Statistical value of rule f is a function:

$$stats_f = (degs_f + icovr_f)$$

where $degs_f$ represents the number of detected degradations by applying the rule f, and $icovr_f$ stands for how many times the coverage was increased by applying rule f. Fuzzer then calculates the number of new mutations for every rule to be applied in four possible ways:

1. The case when $N = 1$, the fuzzer will generate one mutation per each rule. This is a simple heuristic without the usage of statistical data and where all the rules are equivalent.

2. The case when $N = min(stats_f + 1, FLPR)$, the fuzzer will generate mutations proportionally to the statistical value of function (i.e. $stats_f$). More mutation workloads are generated for more successful rules. In case the rule f has not caused any change in coverage or performance (i.e. $stat_f = 0$) yet, the function will ensure the same result as in the first strategy. File Limit Per Rule (FLPR) serves to limit the maximum number of created mutations per rule and is set to value 100.

3. Heuristic that depends on the total number of degradation or coverage increases ($total$). The ratio between $stats_f$ and $total$ determines the probability $prob_f$, i.e. the probability whether the rule f should be applied, as follows:

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

   and we choose N as:

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

   Until some change in coverage or performance occurs, (i.e. while $total = 0$), one new workload is generated by each rule. After some iterations, more successful rules have higher probability, and so they are applied more often. On contrary rules with a poor ratio will be highly ignored. However, since they still may trigger some changes we round them to the probability of 10%.

4. The last heuristic is a 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, but the equation for choosing N is modified to:

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

Our fuzzer uses this method by default because in our experience it guarantees that it will generate enough new workloads and will filter out unsuccessful rules without totally discarding them. In case that target program is prone to workload change and the user wants better interleaving of testing phases, it is recommended to use the third method because the maximum number of all created mutations in one iteration is limited by the number of selected mutation rules.

5.6.3 Gathering Interesting Mutations

We usually run fuzzing for a longer period of time trying to trigger as many changes or faults as possible. To maximise 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, because the estimated instruction path length is not satisfactory, hence we discard this workload. The threshold for discarding mutations is multiple of base coverage, set to 1.5 by default, but it can also be specified by the user. A mutation is classified as an interesting workload in case two criteria are met:

$$cov_{mut} > cov_{threshold} \;\;\&\;\; cov_{mut} > cov_{parent}$$

i.e. it has to exceed the given threshold and achieve a higher number of executed lines than its predecessor.

In addition, we feel that the user may not know the ideal threshold and the default value may be too high or too low. Therefore, the constant which multiplies the base coverage (and thus determines the threshold) changes dynamically during fuzzing. In case it is problematic to reach the specified coverage threshold, the value of the constant decreases and thus gives more chance for further mutations to succeed. Vice versa, if the mutations have no problem to exceed the threshold, the value of the constant is probably too low, and hence we increase it.

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 can be interesting for us as well because an incorrect internal program state can contain some degradation and in case of error, handlers can also contain degradation.

The Final Phase of Iteration

After the mutation, all the interested workloads are collected and ready for real testing to detect performance changes. Testing is done similarly to the initial profile-guided testing (Section 5.4.2), but instead we test with fuzzed interesting workloads. If the Perun detect some performance degradation, the particular mutation's rate is recalculated, fuzzer update its statistics of mutation rules, and one iteration is at the end.

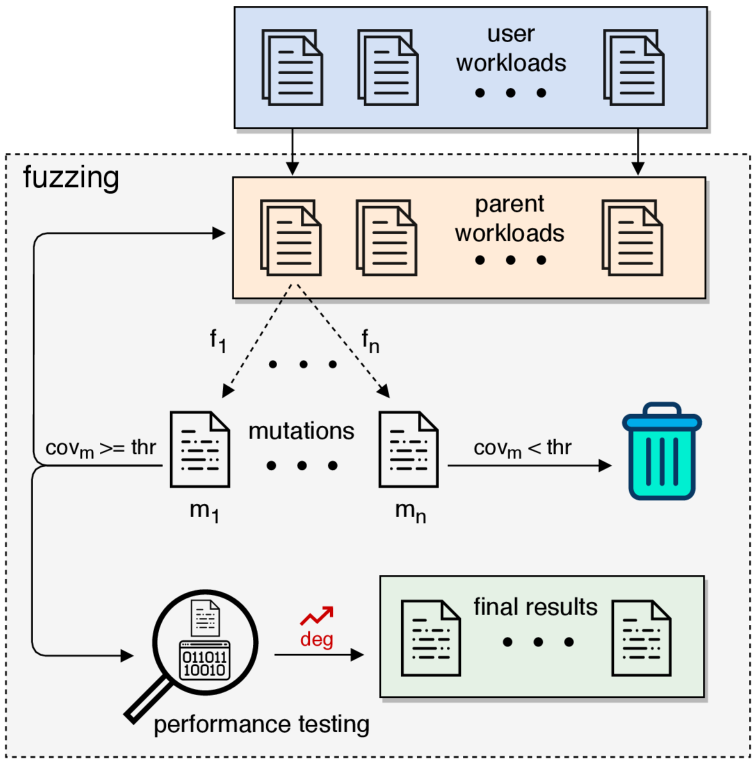

Figure 5.3: Lifetime of workloads, for better understanding. User workloads become parents, we are applying a set of rules on them in order to create their mutations, and the ones with good coverage join the parent set. These workloads may also join the final results set, if they incur a performance degradation.

5.7 Interpretation of Fuzzing Results

During the fuzzing, every file executed with the target program, where the collected runtime data showed a performance drop, joins the set of *final results*. A tester can then analyse these workloads manually. In addition, fuzzer also produces files by which program terminated with an error, or ran too long and these files are stored in specific folders. Other mutations that have been created while running fuzz testing are removed. An example of the structure and content of an output directory is shown in Listing 12.

In order to interpret the results of fuzzing we propose two visualisation techniques: time series and workload difference. The **time series** graphs show the number of found mutations causing degradation and the maximum recorded number of lines executed per one run. From these graphs, one can e.g. read the time needed to achieve sufficient results and estimate orientation time for future testing. In both graphs are denoted three statistically significant values: first quartile, second quartile (median) and third quartile from the y-axis values. The intention is to illustrate at what point in time we have achieved the individual portion of the result.

For plotting time series graphs we used matplotlib[1] library and difflib[2] module helps to calculate deltas between files. Examples of results interpretation are shown in Figures 5.4 and 5.5.

[1]matplotlib — https://matplotlib.org/
[2]difflib — https://docs.python.org/3/library/difflib.html

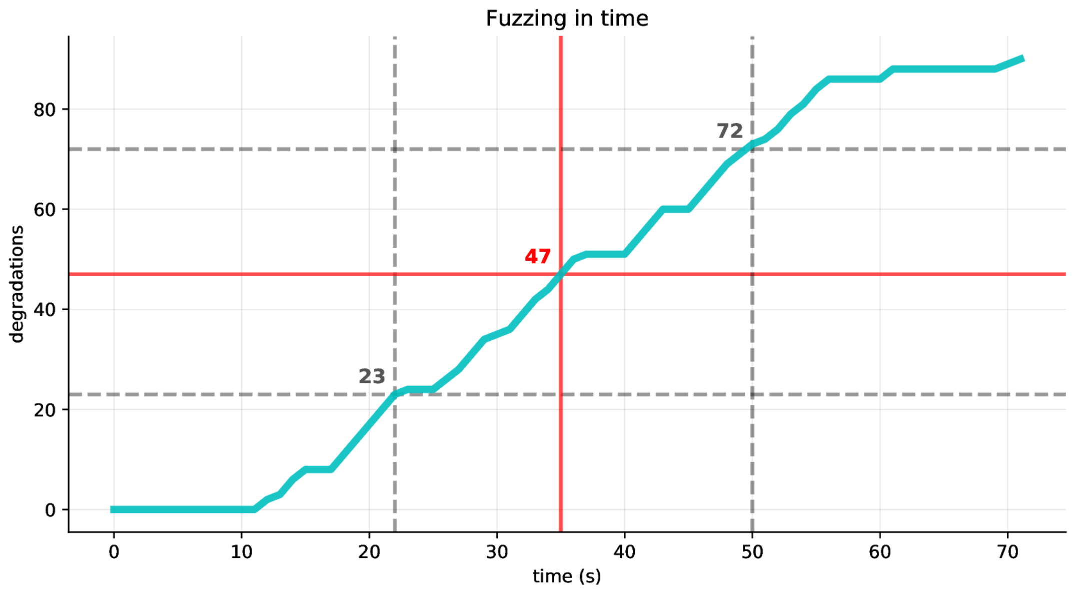

Figure 5.4: Example time series graph, that demonstrates the growth of the detected degradations during the fuzz testing. We can see when the first degradation was detected, when a quarter of the total number of degradations was reached (first quartile, time: 22 s), when approximately half of the total degradations (second quartile, time: 35 s) and three quarters of the total degradations (third quartile, time: 50 s). In the end, the curve stabilised slightly because we found such mutations with which the run of the program took longer.

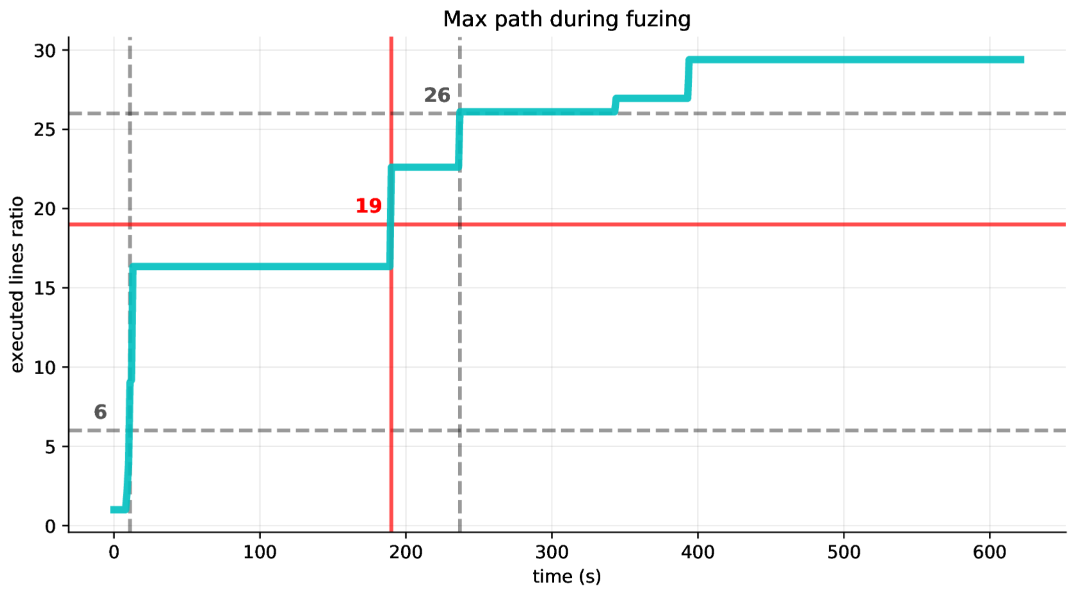

Figure 5.5: Example time series graph, that shows the growth of a maximum number of executed LOC during fuzzing. In first seconds of fuzzing we found a mutation that force target program to execute 16 times more LOC in comparison with the initial seed, and the ratio gradually increases to 29. Three quartiles denote time when 25%, 50% and 75% of the ratios are less than quartile value. User can find raw data of graphs with the exact values in `logs` directory.

Besides visualisation, we create **diff file** for every output file. It shows the differences between files and the original seed, from which the file was created by mutation. The file is in HTML format, and the differences are color-coded for better orientation. Example of diff file is shown in Listing 11.

```
---
+++
@@ -1 +1 @@
-spse1po@gmail.com
+spsepogma.cospsepogma.com
```

Listing 11: Diff file (in uniffied format) shows the differences between the mutated workload and a seed file. Green lines are fuzzed lines that replaced the original content, which is in red color.

```
output/
|--- diffs
     |--- medium_words-02000b239d024dbe933684b6c740512e-diff.html
     |--- medium_words-389d4162ad6641d187dc405000b8d50a-diff.html
     |--- medium_words-39b5d7aa55fd404aa4d31422c6513e2c-diff.html
|--- faults
     |--- medium_words-389d4162ad6641d187dc405000b8d50a.txt
|--- graphs
     |--- coverage_ts.pdf
     |--- degradations_ts.pdf
|--- hangs
     |--- medium_words-39b5d7aa55fd404aa4d31422c6513e2c.txt
|--- logs
     |--- coverage_plot_data.txt
     |--- degradation_plot_data.txt
     |--- results_data.txt
|--- medium_words-02000b239d024dbe933684b6c740512e.txt
```

Listing 12: An example structure of output directory, which is hierarchically divided into five subdirectories: `diffs` contains diff files of the workloads, `faults` includes workloads that led to a fault or crash of the target application, `graphs` contains time series graphs in PDF format, `hangs` contains workloads which forced the program to reach the timeout, and `logs` where are stored raw data used for plotting the graphs and results of fuzzing in plain-text format. On the same level are the workloads denoted as final results, i.e. causing performance degradation.

5.8 Fuzzer Interface

Fuzz unit offers a command line interface (CLI) for interaction with user. To start the fuzzing, one uses command `perun fuzz` with the specification of the tested command, along with arguments, initial workloads, selected collector and possibly postprocessors (with their additional parameters), similar to Listing 3. Moreover, the user can customise the fuzzing process by using additional options, e.g. with a goal to:

- determine the time limit for fuzz testing,
- set the maximum size of generated mutation,
- define the paths to source and `.gcno` files (inevitable for coverage testing),
- set the values of the limitations for coverage-guided testing (line 11 in Listing 6),
- determine the initial value of coverage increase ratio,
- define the maximum time for execution with one workload (timeout for hangs),
- determine the strategy for choosing the number of generated mutations for the rules,
- filter the initial set of workloads by regular expression,
- define custom rules by attaching the YAML file with their specifications.

The user also has to specify the path to the directory where the results of fuzzing will be stored in a hierarchic structure as illustrated in Listing 12. Running `perun fuzz --help` will list all available options that user can specify in order to customise the fuzzing. After fuzzing, summary information about the testing will appear on the standard output as demonstrates Appendix A.

5.9 Testing the Fuzz Unit

Properly tested code with sufficient coverage is also one of the conditions for incorporating into the Perun tool. The test coverage of fuzz unit is listed in Listing 13.

```
Name                                                                Cover
-----------------------------------------------------------------------
perun/fuzz/__init__.py                                               100%
perun/fuzz/coverage.py                                                93%
perun/fuzz/factory.py                                                 90%
perun/fuzz/filesystem.py                                             100%
perun/fuzz/filetype.py                                               100%
perun/fuzz/interpret.py                                              100%
perun/fuzz/methods/binary.py                                         100%
perun/fuzz/methods/textfile.py                                       100%
perun/fuzz/methods/xml.py                                            100%
perun/fuzz/perun_based.py                                             96%
```

Listing 13: Results of unit testing with coverage of individual modules.

Chapter 6

Experimental Evaluation

We tested our performance fuzzer on several case studies to measure its efficiency of generating exhausting mutations. This chapter explores several performance issues in data structures such as hash table or unbalanced binary tree, and a group of regular expressions that have been confirmed as harmful. All the tests ran on a reference machine Lenovo G580 using 4 cores processor Intel Core i3-3110M with maximum frequency 2.40GHz, 4GiB memory, and Ubuntu 18.04.2 LTS operating system.

6.1 Sorting Vulnerabilities

Unbalanced Binary Tree (UBT). Time consumption of inserting to an unbalanced binary tree highly depends on the order of insertion. Even though it is expected to consume $O(n.log(n))$ time when inserting n elements, 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.

First, we constructed files with randomly generated 1 000 integers in the range of <0, 1 000> and 10 000 integers in the range of <0, 10 000>, and we used them as initial seeds ($seed_1$, $seed_2$) to a program that creates an UBT, iteratively inserts elements, and at the end prints the created UBT. We expected that the program performance will highly depend on the amount of workload data, so with the aim to avoid large files we limited the maximum size of mutations.

Table 6.1: The worst-case mutations as workloads for program that manipulates with an UBT. Although our first testing found some workloads reporting degradation, the change was not that impressive. Within the second testing, even a workload about half the size of the initial seed could force the program to create over 125 times deeper binary tree. The worst-case mutation that is as big as original seed, but with sorted elements, prolong the program run more than 100 times (two orders of magnitude degradation). Note that though the worst-case height of UBT when inserting 10 000 elements should be 9 999, the numbers of the initial seed were randomly chosen allowing repeat.

	size [B]	**runtime [s]**	**executed LOC ratio**	**tree height**
$seed_1$	3 879	0.011	1.00	21
$worst\text{-}case_{11}$	1 939	0.033	5.94	309
$worst\text{-}case_{12}$	3 879	0.110	24.46	625
$seed_2$	48 913	0.109	1.00	26
$worst\text{-}case_{21}$	24 456	**2.927**	49.34	3 253
$worst\text{-}case_{22}$	48 912	**11.014**	187.36	6 346

Analysis of worst-case mutations confirmed that unbalanced binary tree degenerates to a linked list when a sorted list is inserted. Table 6.1 presents the results of the program run with the worst-case workloads from each testing. The rules applied on the most exhausting workloads are listed in Table 6.2.

Table 6.2: Table shows the sequence of mutation rules that transformed the seeds into worst-case workloads. Each rule is identified by a label, as defined in Section 4.4. One can see that rules for sorting were more frequently used and thus more successful in mutation.

	used mutation rules
worst-case$_{11}$	[T.7, T.6, T.3, T.6, T.2, T.6]
worst-case$_{12}$	[T.7, T.6, T.1, T.7, T.4]
worst-case$_{21}$	[T.6]
worst-case$_{22}$	[T.7]

std::list + std:find. In our second experiment, we tested the standard library list (`std::list`[1]) which is usually implemented as a doubly-linked list, and we performed a search with `std::find`[2] function. The tested program reads strings from a file, saves them to list and subsequently performs a search for each of them. First initial seed contained 5 000 random english words (*seed*$_1$), and in the second set of tests the initial seed contained 10 000 random english words (*seed*$_2$). For each seed the program run for 266 milliseconds, and 524 milliseconds respectively in average to fill the list and then find every word. In first experiments, we set the maximum size of generated workload to the value of initial seed and in the second one to double of the value. After testing we collect the worst-case workloads and their impact on program is shown in Table 6.3. The rules that led to transformation of the seeds into the worst-case workloads are listed in Table 6.4.

Table 6.3: The most greedy generated workloads compared to initial workload. Processing the worst-case workload, which was the same size as the seed, took program slightly more time to process, roughly in similar proportion as executed LOC ratio. By inspection of these workloads we noticed, that the parts of them are sorted. As one can see, the greater performance change was discovered when the seed containing around 10 000 words, which was expected. Note that worst-case file is two times bigger, contains two times more words, but incur one order of magnitude degradation. In comparison, the execution with a file contained the same words as *worst-case*$_{22}$, but randomly shuffled, took only 2.102 seconds in average.

	size [B]	**runtime [s]**	**executed LOC ratio**	**words**
seed$_1$	37 459	0.266	1.00	5 000
worst-case$_{11}$	37 458	0.485	1.88	5 003
worst-case$_{12}$	74 918	**1.860**	7.53	10 011
seed$_2$	74 915	0.524	1.00	10 000
worst-case$_{21}$	74 897	1.876	3.78	10 041
worst-case$_{22}$	149 830	**7.278**	15.05	20 024

[1]std::list — https://en.cppreference.com/w/cpp/container/list
[2]std::find — https://en.cppreference.com/w/cpp/algorithm/find

Table 6.4: Table lists history of applied rules for worst-case mutations. Notice, that rules providing sort of the elements (T.6 and T.7) appear in the history of every mutation.

	used mutation rules
worst-case$_{11}$	[T.3, T.7, T.6, T.2, T.6, T.6]
worst-case$_{12}$	[T.2, T.7, T.1, T.8, T.1, T.6, T.1]
worst-case$_{21}$	[T.7, T.3, T.3, T.3, T.3, T.3, T.3, T.3, T.3]
worst-case$_{22}$	[T.7, T.1, T.6]

6.2 Regular Expression Denial of Service (ReDoS).

In this case study, we tested artificial programs which use `std::regex_search`[3] with regular expressions inspired by existing ReDoS attacks. ReDoS is an attack based on algorithmic complexity where regular expression are forced to take long time to evaluate, mostly because of backtracking algorithm, and leads to the denial of service.

StackOverflow trim regex. The first experiment is the regular expression that caused an outage of StackOverflow in July, 2016 [2]. An artificial program reads every line and search for match with the regular expression. We used simple source code in C performing parallel grep as an initial seed, written in 150 lines. With only two tests, we could force the vulnerability, as we show in Table 6.5. Which rule is responsible for revealing the weakness can be found in Table 6.6.

Table 6.5: The results from two testings with size limitation set to 5 000 and 10 000 bytes. Analysis of worst-case mutations showed that long sequences of whitespaces not ending with the end of line caused the regex engine to backtrack repeatedly. As one can see, the *worst-case*$_1$ mutation achieved over 16 times longer runtime (one order of magnitude) with only 5 lines of code where whitespaces take 97% of a space. Since we used dynamically collected statistics, fuzzing gave the advantage to whitespace mutation rules, because of their success and therefore they were applied more often. It is more visible in the *worst-case*$_2$, where fuzzer could simply enlarge the file by increasing the number of lines, but instead focused on white characters.

	size [B]	**runtime [s]**	**executed LOC ratio**	**lines**	**whitespaces**
seed	3 535	0.096	1.00	150	306
worst-case$_1$	5 000	**1.566**	24.32	5	4881
worst-case$_2$	10 000	**2.611**	41.38	17	9603

[3]std::regex_search — https://en.cppreference.com/w/cpp/regex/regex_search

Table 6.6: Multiple uses of rule that inserts whitespaces to random position result into big gaps not ending with end of line: the weakness of tested regular expression.

	used mutation rules
worst-case$_1$	[T.10, T.10, T.10, T.10]
worst-case$_2$	[T.10, T.10, T.10, T.10, T.10]

Email validation regex. This regular expression is part of the Regular Expression Library[4] and is marked as malicious and triggering ReDoS. We constructed a program that takes an email address from a file and tries to find a match with this regular expression. As an initial seed we used a file containing valid email address 'spse1po@gmail.com'. We ran two tests, in the first case with an email that must contain the same count of characters as the seed, and in the second case it can contain twice the size. We present the results in Table 6.7 and rules that were used on these mutations are listed in Table 6.8.

Table 6.7: Worst-case mutations for email validation regex. The longer lines not containing '@' sign cause catastrophic backtracking and were terminated (i.e. the run with them take too much time). Even that the size limit for the second test was set to double of the seed (i.e. 36 bytes), the best result was 25 bytes long malformed workload. The reason is that bigger workloads were: (1) not that properly constructed, or (2) too greedy so program reached the set timeout. Because of that, the fuzzer also reported another 8 mutations classified as hangs, and with one of them (*worst-case*$_{2hang}$) the program terminated after more than 5 hours of running.

	size [B]	**runtime [s]**	**executed LOC ratio**
seed	18	0.016	1.00
worst-case$_1$	18	0.176	70.83
worst-case$_2$	25	**10.098**	4 470.72
worst-case$_{2hang}$	36	**>5 hours**	∞

Table 6.8: Two rules, namely removing random character and extending a size of line, were mostly encouraged in the generation of the presented workloads.

	used mutation rules
worst-case$_1$	[T.15, T.8, T.15, T.1]
worst-case$_2$	[T.15, T.15, T.1]
worst-case$_{2hang}$	[T.15, T.15, T.1]

In the following we list the most greedy workloads from each testing and their **content**:

- *worst-case*$_1$: `spse1pogailcspse1p`
- *worst-case*$_2$: `spse1poailcospse1poailco`
- *worst-case*$_{2hang}$: `spse1poailcospse1poailcospse1poailco`

[4] http://regexlib.com/REDetails.aspx?regexp_id=1757

Java Classname validation regex. This vulnerable regular expression for validation of Java class names appeared in OWASP Validation Regex Repository[5]. The testing program was similar to the previous one: reads a class name from a file and tries to find a match with this regular expression. Initial file had one line with string 'myAwesomeClassName'. To avoid the large lines, first we set a size limit for mutations to the size of the initial seed (19 bytes), then to double and finally to quadruple of the size. We present the results of these three tests in Table 6.9. In addition, Table 6.10 shows the order of rules used to mutate the initial seeds.

Table 6.9: We detected two orders of magnitude degradation within run of program with the worst-case from the last test case (*worst-case*$_3$). The fuzzer generates and stores another 26 files that was classified as hangs. By additional testing we found the *worst-case*$_{3hang}$ workload which had enormous impact on program performance, and program did not terminate even after 13 hours lasting run.

	size [B]	**runtime [s]**	**executed LOC ratio**
seed	19	0.005	1.00
worst-case$_1$	19	0.016	14.31
worst-case$_2$	36	1.587	2 383.99
worst-case$_3$	78	**3.344**	5 056.67
worst-case$_{3hang}$	78	∞	∞

Table 6.10: Table lists the rules in order they was applied on the initial seeds and created malicious workloads. Removing characters together with data duplicating, appending whitespaces and other rules collaborated on generation of the worst-case mutations for this case study.

	used mutation rules
worst-case$_1$	[T.8, T.15, T.8, T.15, T.15, T.1, T.12, T.8, T.1]
worst-case$_2$	[T.8, T.15, T.15, T.2, T.8, T.15]
worst-case$_3$	[T.8, T.15, T.1, T.4, T.2]
worst-case$_{3hang}$	[T.8, T.15, T.1, T.15, T.2]

We again list the **content** of generated mutations:

- *worst-case*$_1$: `mywesomelassamemywm`
- *worst-case*$_2$: `mywesomelassamemywesomelassam␣␣␣␣␣␣␣`
- *worst-case*$_3$: `ssammyAwesomelassammyAweiomelassaVmyAwes×melassammmyAwesome lassammyAweomel`
- *worst-case*$_{3hang}$: `laalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaala alaalaalaalaalaalaal`

We also tested other regular expressions, which can be forced to an unlucky backtracking, e.g., expressions to validate a HTML file, search for a specific expression in CSV files or validation of a person name from OWASP Validation Regex Repository. Some of them are part of the evaluation in an article presented at Excel@FIT'19 conference [11].

[5] https://www.owasp.org/index.php/OWASP_Validation_Regex_Repository

6.3 Hash Collisions

Finally, we tried our fuzzer on a simple word frequency counting program, which uses hash table with a fixed number of buckets (12 289 exactly) and the maximum length of the word limited to 127. The distribution of the words in the table is ensured by the hash function. It computes a hash, which is then used as an index to the table. Java 1.1 string library used a hash function that only examined 8-9 evenly spaced characters, which can result into collisions for long strings [19]. We have implemented this behaviour into an artificial program. The likely intention of the developers was to save the function from going through the whole string if it is longer. Therefore, for fuzzing, we initially generated a seed with 10 000 words of 20 characters and started fuzzing. To compare the results we chose the DJB hash function[6], as one of the most efficient hash functions. Tables 6.11 and 6.12 show the result of this last experiment.

Table 6.11: After only 10 minutes of fuzzing each test case was able to find interesting mutations. We then compared the run by replacing the hash function in early Java version with DJB hash function, which computes hash from every character of a string. Table shows, that worst-case workloads have much more impact on performance of the hash table and less stable times using Java hash function, compared to DJB. With such a simple fuzz testing developers could avoid similar implementation bugs.

		Java 1.1 hash function		**DJB hash function**	
	size [kB]	**runtime [ms]**	**LOC ratio**	**runtime [ms]**	**LOC ratio**
seed	210	26	1.0	13	1.0
worst-case$_1$	458	**115**	3.48	**27**	2.19
worst-case$_2$	979	**187**	7.88	**43**	4.12

Table 6.12: Table shows the sequence of mutation rules that transformed the seed into worst-case workloads. In this experiment the rules that duplicates data (T.2), increases number of lines (T.3), changes and removes random characters (T.4 and T.15) were the most frequent.

	used mutation rules
worst-case$_1$	[T.2, T.3, T.15, T.15, T.11, T.15]
worst-case$_2$	[T.2, T.3, T.4, T.15, T.9, T.4, T.2, T.3, T.15, T.15]

We also tried our solution on projects that worked with binary and XML files. Since they did not incur any changes in performance, they are not part of the experimental evaluation. Therefore, improving the existing binary and domain-specific rules together with designing new ones is one of our future goals.

[6]http://www.partow.net/programming/hashfunctions/#DJBHashFunction

Chapter 7

Conclusion

In this thesis, we introduced a fuzzing machine generating malicious inputs focusing on performance weaknesses. We use specific methods to mutate the files, dynamically analyse their efficiency, collect coverage information and use the Perun tool to measure information of program run. Moreover, after fuzzing we provide information about testing in raw and graphical form, and store the worst-case inputs along with their deltas against the original file. Our solution revealed weaknesses in artificial projects working with various data structures and harmful regular expressions, and their performance extremely degraded with processing mutated inputs.

In desing of mutation rules we still mainly focus on text files, hence our future work will focus mainly on proposing more performance tuned rules for, e.g., binary files or other domain-specific types of files. Moreover, we want to add 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.

Bibliography

[1] gcov—a Test Coverage Program. [Online; visited 4.5.2019].
Retrieved from: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[2] Outage Postmortem - July 20, 2016. [Online; visited 4.5.2019].
Retrieved from: https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016

[3] When does the worst case of Quicksort occur? [Online; visited 4.5.2019].
Retrieved from: https://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/

[4] Functional Testing Vs Performance Testing: Should It Be Done Simultaneously? April 2019. [Online; visited 4.5.2019].
Retrieved from: https://www.softwaretestinghelp.com/functional-testing-and-performance-testing/

[5] Clarke, T.: 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.
Retrieved from: https://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-04.pdf

[6] Edholm, E.; Göransson, D.: *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.
Retrieved from: http://publications.lib.chalmers.se/records/fulltext/238600/238600.pdf

[7] Evron, G.: Fuzzing in the Corporate World. December 2006. presentation, 23rd Chaos Communication Congress.
Retrieved from: https://fahrplan.events.ccc.de/congress/2006/Fahrplan/attachments/1248-FuzzingtheCorporateWorld.pdf

[8] Fiedor, T.: Perun: Performance Version System. [Online; visited 4.5.2019].
Retrieved from: https://github.com/tfiedor/perun

[9] Grzybowská, M.: *Graphical User Interface for Performance Control System*. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. 2018.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=19093

[10] Lemieux, C.; Padhye, R.; Sen, K.; et al.: PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. New York, NY, USA: ACM. 2018. ISBN 978-1-4503-5699-2. pp. 254–265. doi:10.1145/3213846.3213874.
Retrieved from: http://doi.acm.org/10.1145/3213846.3213874

[11] Liščinský, M.: Fuzz testing of program performance. In *Excel@FIT'19*. Brno, Czech Republic. 2019.

[12] Machiraju, S.; Gaurav, S.: *Hardening Azure Applications: Techniques and Principles for Building Large-Scale, Mission-Critical Applications*. Apress. 2018. ISBN 9781484241882.

[13] McNally, R.; Yiu, K.; Grove, D.; et al.: Fuzzing: The State of the Art. Technical Report DSTO–TN–1043. Defence Science and Technology Organisation. Edinburgh, South Australia 5111, Australia. 2012.
Retrieved from: http://www.dtic.mil/dtic/tr/fulltext/u2/a558209.pdf

[14] Miller, B. P.; Fredriksen, L.; So, B.: An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*. vol. 33, no. 12. December 1990: pp. 32–44. ISSN 0001-0782. doi:10.1145/96267.96279.
Retrieved from: http://doi.acm.org/10.1145/96267.96279

[15] Molyneaux, I.: *The Art of Application Performance Testing, 2nd Edition*. O'Reilly Media, Inc.. 2014. ISBN 9781491900536.

[16] Myers, G. J.; Sandler, C.: *The Art of Software Testing, Second Edition*. John Wiley & Sons, Inc.. second edition. 2004. ISBN 0471469122.

[17] Nadgir, P.: Performance Testing : An Overview.
Retrieved from:
http://hisolve.com/images/Performance_Testing_White_Paper.pdf

[18] Pavela, J.; Stupinský, Š.: Towards the detection of performance degradation. In *Excel@FIT'18*. Brno, Czech Republic. 2018.

[19] Sedgewick, R. W. K.: Hashing. 2006. presentation, Computer Science Department at Princeton University.
Retrieved from: https://www.cs.princeton.edu/courses/archive/fall06/cos226/lectures/hash.pdf

[20] Serebryany, P., K.; Collingbourne: Beyond Sanitizers: Guided fuzzing and security hardening. October 2015. presentation, LLVM Developer's Meeting.
Retrieved from: https://llvm.org/devmtg/2015-10/slides/SerebryanyCollingbourne-BeyondSanitizers.pdf

[21] Stupinský, Š.: *Automatic Detection of Performance Degradation*. Project practise. Brno University of Technology, Faculty of Information Technology. 2018.

[22] Sutton, M.; Greene, A.; Amini, P.: *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional. 2007. ISBN 0321446119.

[23] Takanen, A.; DeMott, J.; Miller, C.: *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, Inc.. first edition. 2008. ISBN 1596932147, 9781596932142.

[24] West, B.; Wengelin, M.: *Effectiveness of fuzz testing high-security applications*. PhD. Thesis. KTH, School of Computer Science and Communication (CSC). 2017.
Retrieved from:
http://www.diva-portal.se/smash/get/diva2:1105827/FULLTEXT01.pdf

[25] Zalewski, M.: American Fuzzy Lop. [Online; visited 4.5.2019].
Retrieved from: http://lcamtuf.coredump.cx/afl/

[26] Zalewski, M.: Binary fuzzing strategies: what works, what doesn't. August 2014. [Online; visited 4.5.2019].
Retrieved from: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html

Appendix A

Example Output of the Fuzzing

```
Fuzzing successfully finished.
Plotting graphs ...
Computing deltas ...
Saving log files ...
Removing remaining mutations ...
================================ RESULTS ================================
Fuzzing time: 611.35s
Coverage testing: True
Program executions for coverage testing: 870
Program executions for performance testing: 300
Total program tests: 1170
Maximum coverage ratio: 3.4898473442156073
Founded degradation mutations: 299
Hangs: 0
Faults: 0
Worst-case mutation: /home/matus/long_words-e99eed7fe20ef2e4e717676a2.txt
============================= MUTATION RULES ============================
id    Caused deg | cov incr    Desription
0     92 times                 Change random characters at random places
1     66 times                 Insert whitespaces at random places
2     114 times                Divide random line
3     6 times                  Double the size of random line
4     72 times                 Append WS at the end of the line
5     0 times                  Remove WS of random line
6     76 times                 Multiplicate WS of random line
7     64 times                 Prepend WS to random line
8     12 times                 Duplicate random line
9     8 times                  Sort words of random line
10    7 times                  Reversely sort words of random line
11    0 times                  Multiplicate word of random line
12    0 times                  Remove random line
13    0 times                  Remove random word of line
14    82 times                 Remove random character of line
```

Appendix B

Storage Medium

`/perun/*` — source code of Perun containing fuzz unit

`/README.txt` — useful information about the storage medium content

`/text/*` — source code of this thesis

`/xlisci02.pdf` — final version of this thesis

`/experiments/*` — source code of experimental projects