BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PERFORMANCE ANALYSIS BASED ON NOISE INJECTION

VÝKONNOSTNÍ ANALÝZA PROGRAMŮ ZALOŽENÁ NA VKLÁDÁNÍ ŠUMU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATÚŠ LIŠČINSKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ FIEDOR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

Student: **Liščinský Matúš, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Software Verification and Testing
Title: **Performance Analysis Based on Noise Injection**
Category: Software analysis and testing
Assignment:

1. Get acquainted with the Perun project. Study recent techniques for performance testing of applications (fuzz-testing, causal profiling, performance amplification),
2. Get acquainted with frameworks for dynamic instrumentation of programs (eBPF, SystemTap, Pin) and their capabilities of noise injection.
3. Propose an evolution algorithm, that will automatically inject noise into analysed programs, evaluate its impact and adjust the amount of injected noise in next iterations. Implement the algorithm as a module in Perun.
4. Propose a visualization or interpretation, that will illustrate the results of the performance blowing process.
5. Demonstrate the solution on (at least) two non-trivial case studies.

Recommended literature:

- *Charlie Curtsinger and Emery D. Berger*. Coz: finding code that counts with causal profiling. In Proc. of SOSP '15.
- *Lu Fang, Liang Dou, and Guoqing Xu*. PERFBLOWER: Quickly Detecting Memory-Related Performance Problems via Amplification. In Proc. of Ecoop'15.
- Official site of the Perun project: https://github.com/tfiedor/perun

Requirements for the semestral defence:

- Items 1 and 2, and at least some development falling under item 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Fiedor Tomáš, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: November 11, 2020

Abstract

In this work, we proposed a Perun-Blower framework which utilises the perfblowing technique: injecting of noise into the functions of the tested program, followed by collecting of runtime data of these functions from the program run and evaluating the impact of the noise on the program performance. We build on the dynamic binary instrumentation of the Pin framework to inject the noise into program. We then focus on finding functions with high impact on performance as well as estimate the thread run's potential acceleration when optimising the particular functions. Moreover, we have extended the existing Trace collector used in the Perun framework to collect the runtime of functions with a new so-called engine based on the Pin framework. We tested the functionality of our implementation on two non-trivial projects, where we were able to find functions (1) with considerable impact on performance, (2) with the most significant optimisation benefit, and (3) whose degradation forces the non-termination of the program after several hours of running.

Abstrakt

Táto práca predstavuje nástroj Perun-Blower, využívajúci perfblowing techniku: vkladanie šumu do funkcií testovaného programu a nasledovné vyhodnotnie vplyvu šumu na výkon programu na základe zozbieraných časových údajov týchto funkcií z behu programu. Implementácia je postavená na dynamickej binárnej inštrumentácii nástroja Pin. Zameriavame sa na hľadanie funkcií, ktoré majú vysoký vplyv na výkon a rovnako tak aj odhad potenciálneho zrýchlenia behu vlákna pri optimalizácii konkrétnej funkcie. Naviac sme rozšírili existujúci Trace collector používaný v nástroji Perun na zbieranie časových dát funkcií, o nový tzv. engine, ktorý je založený práve na nástroji Pin. Funkčnosť implementácie sme otestovali na dvoch netriviálnych projektoch, kde sme dokázali nájsť funkcie (1) so značným vplyvom na výkon, (2) s najvýznamnejším optimalizačným prínosom a (3) funkcie, ktorých degradácia spôsobí, že vykonávanie programu sa neskončí ani po niekoľkých hodinách.

Keywords

causal profiling, performance testing, virtual amplification, noise injection, performance bottlenecks, optimisation

Kľúčové slová

kauzálne profilovanie, výkonnostné testovanie, virtuálna amplifikácia, vkladanie šumu, výkonnostne úzke miesta, optimalizácia

Reference

LIŠČINSKÝ, Matúš. *Performance Analysis Based on Noise Injection.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Fiedor, Ph.D.

Rozšírený abstrakt

Každý softvérový projekt by si mal udržiavať nielen svoju funkčnosť, ale aj primeranú výkonnosť. Používanie nástroja *Perun* [23] je jedným z riešení pre správu a pravidelnú analýzu kvality výkonu softvérového projektu. Perun ponúka profilovanie výkonu a spája rôzne štatistické údaje o výkone s konkrétnymi verziami projektu vďaka interakcii so systémami VCS, ktoré umožňujú sledovať zmeny v projektoch počas vývoja softvéru.

Detekovať výkonnostné chyby v softvéri nie je ľahká úloha. Na rozdiel od funkcionálnych chýb, je hľadanie problémov s výkonom obtiažnejšie, pretože tieto chyby sa zvyčajne prejavia iba za určitých podmienok. Pomerne často sa v rámci celého programu vyskytuje niekoľko menších problémov s výkonom: zdanlivo neškodné výkonnostné problémy, ktoré sa prejavujú pri veľkých alebo špecificky štruktúrovaných vstupoch, vedúcich k neschopnosti programu dosiahnuť očakávaný výkon. Preto je potrebné tieto nenápadné problémy "nafúknuť" tak, aby mohli byť detekované.

Snaha nájsť problémy s výkonom, na ktoré majú veľký vplyv vstupy, je už implementovaná v jednotke Perun-fuzz v nástroji Perun, využívajúca *fuzz testovanie* (automatické generovanie neočakávaných alebo náhodných dát ako vstupov, poskytované programu). Fuzz testovanie však nemusí skrytú výkonnostnú chybu odhaliť. Táto práca sa preto nezameriava na zmenu vstupov programu, ale na zmenu samotného programu. Naša predstava je, že na prejavenie výkonnostnej chyby použijeme techniku perfblowingu: umelého nafukovania výkonu programu pomocou vkladania šumu. Prichádzame tak s myšlienkou opakovaných perfblowing experimentov s injektovaním určitého množstva šumu do programu.

Navrhovaný perfblowing algoritmus sa zameriava na množinu funkcií získaných z testovaného binárneho programu, vybraných podľa určených pravidiel (napr. počet volaní funkcie alebo priemerný čas trvania funkcie). Tieto funkcie tvoria takzvaný *corpus*, t.j. množina kandidátnych funkcií pre perfblowing experimenty. Podobne ako v prípade evolučných algoritmov, každá funkcia má priradené *fitness* skóre, ktoré určuje kvalitu funkcie s ohľadom na cieľ perfblowingu: (1) nájdenie výkonnostne úzkych miest vložením šumu do jednej funkcie alebo (2) odhad zrýchlenia programu (v prípade, že by sme optimalizovali funkciu) vložením šumu do všetkých okrem vybranej funkcie.

Každý experiment vloženia šumu má určitý vplyv na výkon; na automatické vyhodnotenie tohto vplyvu je však potrebné zbierať údaje za behu experimentu. Zbieranie údajov za behu a vkladanie šumu vyžadujú zásah do testovaného programu. Nástroj *Pin* [8] implementuje techniku vkladania kódu do existujúceho programu pomocou dynamickej binárnej inštrumentácie (DBI). V porovnaní s nástrojmi na DBI, ktoré sú už zapojené v nástroji Perun, Pin ponúka viac možností inštrumentácie a analýzy kódu.

Zozbierané údaje z experimentu ovplyvňujú fitness hodnotu funkcie, ktorá bola vybraná pre experiment. Vďaka tomu dosiahneme kontrolovanejší výber funkcií v nasledujúcich iteráciách. Výsledky každého uskutočneného experimentu sú uložené s cieľom poskytnúť používateľovi ich interpretáciu v nasledujúcich formách: (1) tabuľka funkcií s najvyšším skóre, (2) graf časovej osi behu najlepšie hodnoteného experimentu, (3) kauzálne profily označujúce vplyv "nafúknutia" na celkovú dobu programu alebo jeho vlákna.

Implemetáciu sme vyhodnotili na dvoch netriviálnych prípadových štúdiách (knižnici `google/re2` pre regulárne výrazy a nástroji `Z3` pre overovanie teorémov). Výsledky ukazujú, že s našou implementáciou dokážeme nájsť: (1) potenciálne výkonnostné úzke miesta (t.j. funkcie, ktoré majú veľký vplyv na celkovú výkonnosť programu) a (2) funkcie, ktoré predstavujú príležitosť na optimalizáciu (a správne odhadli zrýchlenie v prípade ich potenciálnej optimalizácie). Počas testovania na `Z3` sme navyše našli niekoľko funkcií, pri ktorých injektovaný šum viedol k uviaznutiu alebo prejavu možnej výkonnostnej chyby.

Performance Analysis Based on Noise Injection

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Tomáš Fiedor, Ph.D. The supplementary information was provided by Ing. Jiří Pavela. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Matúš Liščinský
May 25, 2021

Acknowledgements

I would like to thank Ing. Tomáš Fiedor, Ph.D., for supervising this thesis, for his patience, constructive approach, and willingness to help me. Furthermore, I would like to express thanks to Ing. Jiří Pavela for his professional help and readiness to answer all my questions.

Contents

Chapter 1

Introduction

> "*Any optimisation that is not about the* ***bottleneck*** *is an illusion of improvement.*"
>
> — Federico Toledo

Developers should make an effort to ship software with performance as efficient as possible. Hence, each project should maintain not only its functionality but also adequate performance. Using the *Perun* [23] tool is one of the solutions for managing and regularly analysing the performance quality of a software project. Perun offers performance profiling and links various performance statistics to specific versions of the project by interaction with *Version Control Systems*[1].

However, it is particularly tricky to detect, e.g., performance fluctuations in software. In contrast to functional bugs, finding performance issues is often difficult because they usually manifest only under certain conditions. Quite often, there are several minor performance issues across the whole program: ostensibly harmless performance problems that manifest under large or specifically structured workloads leading to the program's inability to achieve the expected performance. Therefore, one needs to inflate these unnoticed problems so that they will be visible and detectable.

The effort to find such performance issues that are greatly influenced by particular inputs is already implemented in the Perun tool leveraging the *fuzz testing* (a software testing technique that involves the automated generation of invalid, unexpected, or random data as inputs, which are provided to a computer program). However, the fuzz testing might miss many cases of performance issues or optimisation opportunities.

Instead, this work does not focus on modification of program workloads but changing the program itself. The idea is to enforce the manifestation of performance issues by utilising perfblowing technique: artificial inflating of program performance based on noise injection. We propose the idea of repeated perfblowing experiments of injecting a certain amount of noise into a program.

The proposed perfblowing algorithm focuses on the set of program functions yielded from the tested program binary and selected according to specified rules (e.g. the number of function calls or the average function runtime). These functions form the so-called corpus, i.e. function candidates for perfblowing experiments. Similar to evolutionary algorithms, the *fitness* score represents each function, determining the quality of function with respect

[1] systems that allow keeping track of the changes in software development projects

to the perfblowing goal: (1) finding performance bottlenecks by injecting the noise into selected function, or (2) estimating the potential speed-up (in the case we would optimise the function) by injecting the noise into all except selected function.

Each injection experiment has some effect on performance; however, to automatically quantify its impact, it is necessary to collect runtime data for the run of the experiment. Nevertheless, the runtime data collecting and the noise injection require modification of the tested program[2]. *Pin* [8] framework implements a technique of inserting the code into the existing program by a dynamic binary instrumentation (DBI) approach. Compared to the frameworks for DBI that are already utilised in Perun, Pin offers more instrumentation and code analysis options.

The collected runtime data of the experiment then affects the fitness value of functions selected for the experiment. This approach leads to more controlled function selection in the next iterations. The results of each performed perfblowing experiment is finally interpreted to a user as follows: (1) as a table of the top-scored functions, (2) as a timeline graph of the best-rated perfblowing experiment run, and (3) as causal profiles denoting the impact of perfblowing experiment on the overall runtime of a program or a program thread.

We evaluated the implementation on two non-trivial case studies (`google/re2` library and `Z3` theorem prover). The results show that we can find: (1) potential bottleneck functions (i.e. the functions that have a big impact on the overall performance of the program) and (2) functions that represent the optimisation opportunity. Moreover, during the testing on `Z3` we found several functions where the injected noise led to a possible performance issue or a deadlock.

Structure of the Thesis. Chapter 2 introduces the topic of enforcing the manifestation of performance bottlenecks using *(performance) fuzz testing* and *performance amplification*. In Chapter 3, we describe available DBI frameworks and their usability for noise injection. We identify the functional and non-functional requirements of this work in Chapter 4. The implementation part of the work begins in Chapter 5 with the description of extending the *Tracer* by a new engine, based on *Pin*. Chapter 6 is the core of this work since it includes the description of the design and implementation of the Perun-Blower framework. Chapter 7 presents the experimental evaluation results, and the final Chapter 8 summarises the achieved results and outlines improvements that could be addressed in future work.

[2] this work focuses on programs in the form of executable binary files

Chapter 2

Enforcing Manifestation of Bottlenecks

We define the performance bottleneck as a location in the program, that significantly slows the overall performance. Thus, considering a program with several modules, its performance is directly related to its slowest unit's performance. We will limit ourselves to analysing performance of programs compilable to binary executable.

Fixing the bottleneck precedes identifying and analysing the underperforming location. The crucial bottlenecks can then be optimised to achieve better performance. However, the performance bottleneck can be tough to identify. In practice, we measure consequences of bottlenecks [6]:

- CPU utilisation: the processor is so busy, it is unable to perform the tasks demanded of it in a timely manner;
- Memory utilisation: the system does not have sufficient or fast enough memory;
- Network utilisation: the communication between two devices suffers from lack of the necessary bandwidth or processing power to complete a task;
- Disk usage: long term storage is the slowest unit in the processing chain, often an unavoidable bottleneck;
- Software limitation: performance decline is caused by the software itself, e.g. inefficient use of allocated resources, poorly designed code structure, etc.

The issues on the software side are resolved by rewriting and patching certain code segments. Thus, we focus on identifying them by enforcing their manifestation. We can divide the approaches to manifestation of bottlenecks into the two following areas.

Derivation of resource intensive workloads. The idea is to use fuzz testing to detect performance deficiencies connected with the *System Under Test (SUT)* and a specific workload. This principle is used by several fuzz testing tools such as PerfFuzz [31], Badger [36], or Perun-fuzz [33].

Amplification of bottleneck effects on performance. The idea is to attempt to highlight the consequences of the not evident performance bottlenecks. This principle was used to find memory-related performance issues in PerfBlower [21], or to identify candidates for virtual speedup in Coz [15].

2.1 Fuzz Testing

Fuzzing (fuzz testing) is a form of a fault injection stress testing, where a range of malformed inputs are send to a software application while monitoring for potential failures [13]. Fuzzers are cheap to deploy for projects that work with simple input interface (e.g. reading from a file given as a command line option), do not suffer from false positives (it is highly unlikely that discovered flaw would be a false positive), but on the other hand monitors only crashes and fails. Typically, fuzz testers are implemented as an automated loop that repeatedly calls SUT with random input strings as command line switches, program inputs, or environment variables. Program crashes or hangs are reported.

Fuzzing can be considered as a particular type of dynamic testing. Fuzzers are simply used to automate calling the SUT with particular inputs. Many people commonly associate fuzzers with detecting buffer overflow, however, advanced and custom fuzzers can do more than simply provide tremendous volume of input to an application. Recently, fuzzers have been used to uncover much more complex flaws than the traditional buffer overflow.

The first fuzz testing approaches were purely based on randomly generated test data (random fuzzing). The research has led to more advanced fuzzing techniques such as mutation-based fuzzing, generation-based fuzzing, or grey-box fuzzing. Grey-box fuzzers can combine previous techniques and may also utilise techniques from other program analysis fields, such as symbolic execution [35].

Random fuzzing is the simplest fuzz testing technique: a stream of random input data is (in a black-box scenario) sent to the SUT. The input data can be sent as, e.g. command line options, events, or protocol packets. This type of fuzzing is, in particular, useful to test how SUT reacts on large or invalid input data. While random fuzzing can find serious vulnerabilities, modern fuzzers use more detailed understanding of the input format that is expected by SUT [35].

Fuzz testing typically includes the following 6 steps:

1. **Identify the target**: choosing the target application which will be tested (e.g. the *vim* text editor);
2. **Identify the inputs**: determining what inputs the target application accepts (e.g. text files or binary data);
3. **Generate fuzzed data**: creating new input data (e.g. by mutating input corpus or generating new inputs from template);
4. **Execute fuzzed data**: feeding the target application with newly generated input (e.g. open the generated workload with *vim*);
5. **Monitor for exceptions**: monitor the target application for interesting behaviour; (e.g. observe that *vim* exited with segmentation fault);
6. **Determine exploitability**: analysing the behaviour and classifying the input. (e.g. determine that the segmentation fault can lead to exploit);

Fuzzers can be divided according to how they generate new inputs. Fuzz data can be generated using predetermined values (e.g. IPv4 address), mutating existing data starting from some input corpus of samples (e.g. bit swaps) or generating new data from scratch (e.g. based on some template or grammar). In particular, we will list 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 given template or grammar specification, which define the structure of the input file that is processed by the target program.

Such template should be accurate, detailed and include all possible options for every field of a structure. This ensures that the fuzzer generates valid data for control fields such as checksums or challenge-response messages and thereby achieves high coverage of program paths. The example template (Peach Pit) used by *Peach* framework is shown in Listing 1. However, creating a bullet-proof template tends to be time-consuming and complex.

```
1  <DataModel name="Chunk">
2    <Number name="Length" size="32" >
3      <Relation type="size" of="Data" />
4    </Number>
5    <Block name="TypeData">
6      <Blob name="Type" length="4" />
7      <Blob name="Data" />
8    </Block>
9    <Number name="crc" size="32" >
10     <Fixup class="Crc32Fixup">
11       <Param name="ref" value="TypeData"/>
12     </Fixup>
13   </Number>
14 </DataModel>
15 ...
16 <DataModel name="Chunk_tRNS" ref="Chunk">
17   <Block name="TypeData">
18     <String name="Type" value="tRNS" />
19     <Blob name="Data" />
20   </Block>
21 </DataModel>
22 <DataModel name="PNG">
23   <Number name="Sig" value="89504e..." />
24   <Block name="IHDR" ref="Chunk_IHDR"/>
25   <Choice name="Chunks" maxOccurs="30000">
26     <Block name="PLTE" ref="Chunk_PLTE"/>
27     ...
28     <Block name="tRNS" ref="Chunk_tRNS"/>
29     <Block name="IDAT" ref="Chunk_IDAT"/>
30   </Choice>
31   <Block name="IEND" ref="Chunk_IEND"/>
32 </DataModel>
```

Listing 1: PNG input model as Peach Pit, which allows to specify PNG file format in *Peach* framework. First, the model specifies the generic data chunk (lines 1-14) with data fields all other data chunks inherit (lines 15-21). The whole PNG file is specified last, at lines 22-32. This model was taken from [40].

The generative method is usually used for simple models or protocols where a construction of a template has no significant cost. Although many of the applications work 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 [18].

The first representative of such a fuzzer was *PROTOS* [42]. Further we can list in chronological order, e.g. *SPIKE* [10], *PEACH* [17], *GWF* [25], *MoWF* [40], or *Skyfire* [43], released in 2017.

Mutational Fuzzer

Mutational fuzzer does not require any specification of input format. Instead, it is based on a set of sample inputs (note that even one single sample file suffices). New workloads are then generated by applying mutating strategies on these initial inputs, so called the *seeds*. In Listing 2 is shown example of such mutation rule for transforming one input line.

```
1  def remove_ws(line):
2    '''Mutation rule: removes whitespaces of the given line
3
4    :param str line: line to be mutated
5    :return str: mutated line (without whitespaces)
6    '''
7    return ''.join(line.split())
8
9  line = "The quick brown fox."
10 print(remove_ws(line))
```

Listing 2: Example of mutation rule that removes all whitespaces of a given line, implemented in `Python`. The rule returns transformed string with no whitespaces: *"Thequickbrownfox."*.

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

The input (either seed or mutation) is valuable and is reused for further work or discarded based on selected factors, e.g. achieved coverage, discovered new program paths, triggered exceptions etc.

Unfortunately, the drawback is that the SUT may reject the mutated input at the beginning of processing of the data during the validation phase since mutations can generate invalid format. Nevertheless, even invalid inputs can sometimes lead to an interesting responses from SUT.

Over the years, many fuzzers leveraging the mutational approach arise, e.g. *AFL* [46], *zzuf* [30], its extension *CERT Basic Fuzzing Framework (BFF)* [28], *BuzzFuzz* [24], *honggfuzz* [26] and many others.

2.1.1 Performance Fuzz Testing (PerfFuzz)

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

PerfFuzz [31] is a (i) coverage-guided, (ii) mutational, (iii) feedback-directed fuzzing engine that uses multidimensional feedback in the AFL's CFG (Control Flow Graph[1]) method. The feedback includes code coverage information (e.g. which CFG edges were executed) and also metrics associated with the program components of interest (e.g. how many times each CFG edge was executed). Additionally, it creates a *performance map* to improve future usability estimation of tested input: performance map is a function $perfmap : K \rightarrow V$, where K is a set of keys corresponding to program components (CFG edges) and V is a set of ordered values (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.

The comparison with AFL shows that AFL initially may find hot spots with higher execution count, but in the end PerfFuzz always managed to find a hot spot with at least twice higher execution count as AFL. Overall, PerfFuzz finds hot spots with over 2× - 18× higher execution counts after 6 hours of lasting experiments [31].

Summary. PerfFuzz is evaluated on four real-world C programs typically used in the fuzzing literature[2] and outperforms previous work (SlowFuzz [39]) by generating inputs that exercise the most-hit program branch 5× to 69× times. However, we note that more performed operations/instructions do not necessarily mean it will cause notable performance difference. PerfFuzz uses so-called blind mutation methods (e.g. flipping random bits, moving or deleting blocks of data, etc.), as well as the AFL on which it is built, so despite that the goal of the fuzzing changed, the way of workload mutation remains the same.

2.1.2 Perun Performance Fuzzer (Perun-fuzz)

Our work is intended to be a part of the Perun tool: an open source lightweight Performance Version System for continuous performance monitoring [23]. Perun-fuzz [33] is a fuzzer incorporated within Perun, generating malicious inputs focusing on performance weaknesses.

Original Technique

Perun-fuzz is based on the mutation based approach, accepting the set of sample *seed* inputs (or workloads), called *input corpus*. The *seeds* should be valid inputs for the target application, so the application terminates on them and yields expected performance. To trigger a performance change of the target system, the fuzzer systematically launch the SUT with differently malformed inputs. The inputs are several times transformed or modified by the sundry mutation rules.

[1] oriented graph, which describes the mutual caller/callee relationship of individual program subroutines

[2] `libpng-1.6.34`, `libjpeg-turbo-1.5.3`, `zlib-1.2.11`, and `libxml2-2.9.7`.

Contrary to PerfFuzz, Perun-fuzz uses specific mutation rules that were inspired by existing performance issues, dynamically analyse their efficiency, collects coverage information, and then uses Perun to measure selected performance aspects of a program run, using its techniques for detecting performance regressions.

One of the proposed mutation rules is doubling the size of a line. This rule duplicate the content before end line character of the random line, e.g. the input line *"The quick brown fox."* will transform to *"The quick brown fox.The quick brown fox."*. This rule focuses on possible performance issues associated with long lines appearing in files. The inspiration comes from the gedit[3] 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 [33].

Before running the target application with malformed inputs, it is necessary to first determine the *performance baseline*, i.e. the expected performance of the program, to which future testing results will be compared. Initial testing first measures code coverage (number of executed lines of code) while executing each initial seed. After that, the median of measured coverage data is considered as baseline for coverage testing. Second, Perun is run on collected memory, trace, time or complexity resource records with initial seeds resulting in baseline profiles. In essence *performance baseline* is a profile describing performance of program on the given corpus. The integration of fuzzer within Perun framework is illustrated in Figure 2.1.

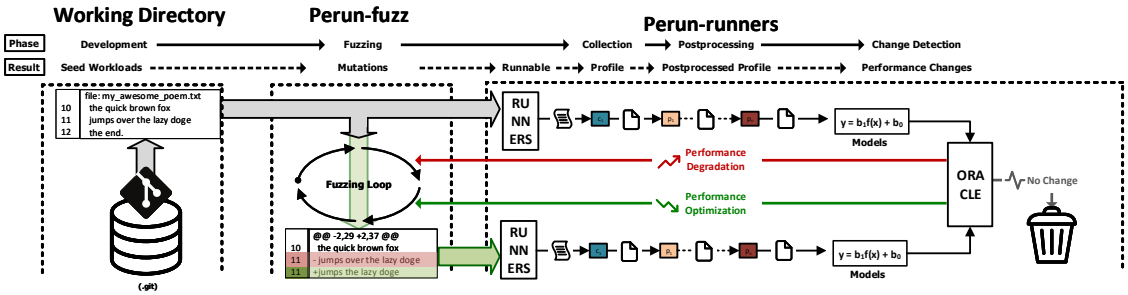

Figure 2.1: Perun-fuzz integrated within Perun framework, adopted from [23]. The fuzzer takes the seed workloads (e.g. problematic workloads for previous version of project) and starts its loop. To evaluate new mutations, it uses the results of analyses yielded from Perun performance testing. If the fuzzer generates the workload which triggers a performance degradation (Perun detects it), the workload is stored. Thus, developers can fix the performance issue and keep the workload for future testing [33].

Every mutation file is tested with the goal to achieve the maximum possible code coverage. The first testing phase consists of gathering the interesting inputs, which increase the number of executed lines compared to input corpus. After gathering the interesting inputs, the fuzzer collects run-time data (memory, trace, and time), transforming the data to a so called target profile and checks for performance changes by comparing newly generated target profile with baseline performance profile [29].

In case that performance degradation has occurred, the responsible mutation file is added to the corpus and therefore can be fuzzed in the future to intentionally trigger

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

more serious performance issues. The intuition to perform coverage testing first is that running coverage testing is faster than collecting performance data (since it introduces certain overhead) and by collecting performance data only newly covered paths will result in more interesting inputs. The process of manipulation with a mutation during fuzzing can be seen in Figure 2.2.

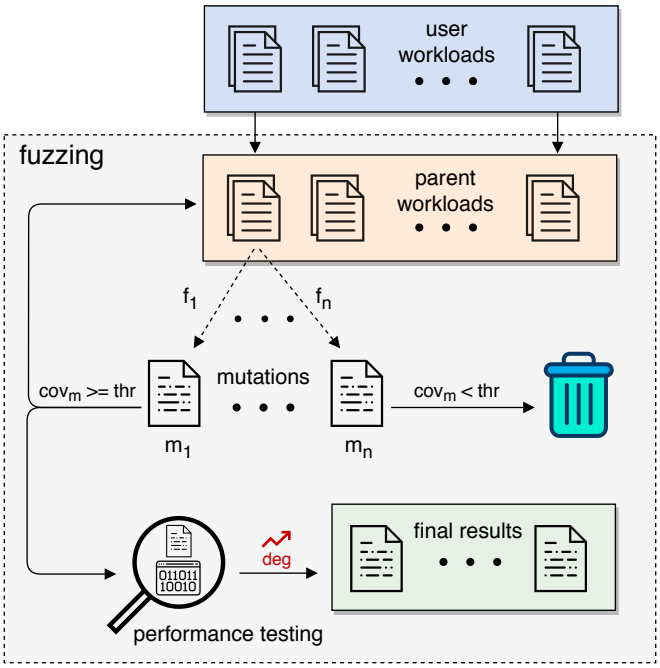

Figure 2.2: Lifetime of workloads: user workloads become parents, we apply a set of rules on them in order to create their mutations, and the ones with good coverage are added to the parent set. These workloads may also join the final results set, if they incur a performance degradation. Workloads that lead to performance degradation are added to final results. Figure taken from [33].

Coverage Analysis Improvement

Unfortunately, the initial version of Perun-fuzz still had its pitfalls. One of them is choosing the so-called interesting mutations in coverage testing, where significant approximation occurs. The sum of executed LOC[4] in the run is used, which is enough for smaller projects, with a smaller number of LOC, but with more robust programs with modules containing several thousand LOC, locations with the local maximum coverage can be lost and the desired change may not be detected. Such cases are quite often in practice, since complex programs contain more complex branching.

The original approach of Perun-fuzz works on the basis of collecting information about code coverage, which is aggregated into one value (number of executed lines of code). With this approximation, the analysis loses accuracy and weakens the ability to detect a significant but local change in code coverage. The aim of the work [32] is to minimise these shortcomings by improving the analysis of coverage when running a program with mutated inputs.

[4]LOC — Lines Of Code

The idea is to use the static call graph of the target program. Coverage data, are then assigned to the call graph structure. In particular, it stores two types of coverage data: *inclusive* and *exclusive* (we describe them below). In addition, the call graph is divided into individual paths from the root node to the leaf nodes. This way the fuzzer works the vectors of inclusive/exclusive coverage of paths, which are compared across the run of the tested program with malformed inputs. This leads to achieving better analysis of the mutated inputs and therefore a better decision whether a mutation is suitable for further testing by one of the selected Perun data collectors (time, memory, and trace).

Exploiting the call graph. Call graph is an oriented graph, which describes the mutual caller/callee relationship of individual program subroutines. Each node represents specific subroutine, and each edge (f, g) indicates the call of subroutine g from within subroutine f. We extend the static call graph with information about code coverage resulting into so called *annotated call graph*, where each node does not only carry information about which function it represents but also its inclusive and exclusive code coverage. In our case, inclusive coverage represents how many lines were executed in a given function during the program run, while the exclusive coverage represents the number of calls of the function only. Each run of the program will not differ from another by the structure of the call graph, however it coverage an change significantly. Unlike the original approach, where the coverage indicator was a single sum, we work with multiple data stored in the graph structure, which offers the chance to more accurately detect and locate eventual changes in code coverage.

Evaluation

The evaluation was conducted on five tests in total, each consisting of fuzzing a certain project for 300 seconds. Each of the projects is artificial with a nested performance bug. Detailed description of the evaluation is described in [33]. The original Perun-fuzz found workloads that prolong the program run 7× to 1000× and increase the number of executed LOC 8× to 5000×, approximately. In the evaluation the fuzzer which exploited the call graph there achieve similar results. However, even when the workload triggered less executed lines than in original approach, the tested projects reached longer runtime, which shows that this approach makes a better focus on the more performance-dependent code areas.

2.2 Causal Profiling

Causal profiling was first introduced by Emery D. Berger and Charlie Curtsinger [14]. Its main motivation was the ability to answer the question developers often would like to know: "How much will optimising this code improve the performance?". Most of the standard profilers only observe the execution of SUT, measure the time spent in functions and how many times functions were called. Hence, the question clearly involves causality; the profilers are not capable to answer such a question.

Thus came the idea of novel profiling technique: causal profiling. Causal profiling aims to indicate where programmers should focus their optimisation efforts and quantify the potential impact of improvement. A causal profiler performs a series of performance experiments during program execution to determine which blocks will cause performance improvement if optimised. The profiler makes a small change to the performance of a single block of code for each experiment. Any impact on overall program performance must be

caused by this change. The rest of this section is based on [14]. The actual causal profiling then selects suitable blocks and perform slowdown and speedup experiments.

Selecting Code Blocks for Experiments. Code that is scarcely executed clearly does not have the potential to impact program performance, hence, optimising it will not have any effect. A causal profiler can select blocks for performance experiments, e.g. using the empirical distribution over the code. Frequently executed blocks are more likely to be selected than infrequently executed blocks. However, frequently executed blocks do not necessarily have to be suitable for optimisation. As an example, we can assume a simple server, that sends a response to a client request whenever it receives one. Meanwhile, the server periodically sends `keep-alive` messages to the client. Optimising the code which produces these messages will not decrease the server's request latency, even though this code would be the most executed overall.

Slowdown Experiments. For each selected block, we want to know if it has significant effect on performance. During this experiment, the profiler inserts a small delay in the subject block each time it is executed. If it keeps the program's performance almost unchanged, the block does not have an performance impact. The larger the detected degradation in the program's performance (with respect to the size of the delay), the more suitable the block can be for optimisation.

"Speedup" Experiments. Even though slowing a particular block can degrade some performance metrics, it does not ensure that optimising the block will improve the performance of the program. For example, if a program performs its work in two threads that take equal amount of time, slowing just one of them will degrade performance but optimising it will not bring any improvement until we optimise both threads. To eliminate these cases of perfectly balanced parallelism, a causal profiler also performs "speedup" experiments. When the subject block is executed, the profiler pauses all other threads. The sum of individual delays is later subtracted from the program's wall running time to compute the effective execution time. The resulting change in performance tells us the expected effect of an optimisation.

2.2.1 Coz

The first causal profiler is Coz [15], the causal profiling that accurately and precisely indicates where programmers should focus their optimisation efforts and quantifies their potential impact. In this implementation, Coz analyses the potential impact of optimising one code line by a specific amount. A line is virtually sped up by inserting a pause to all other threads each time the line is executed.

Virtual speedup. To achieve the effect of the actual speedup, a virtual speedup delays every other thread every time a code line is executed. Coz randomly chooses a code line and the amount of speedup within each experiment. After termination of the application, Coz uses three values in order to calculate the potential speedup of the case that the chosen line was optimised: *original runtime* (t_o), the sum of all *inserted delays* ($\sum d$), and *runtime with virtual speedup* (t_v). The potential speedup (t_s) is then calculated as:

$$t_s = t_o + \sum d - t_v \tag{2.1}$$

The speedup experiment is illustrated in Figure 2.3.

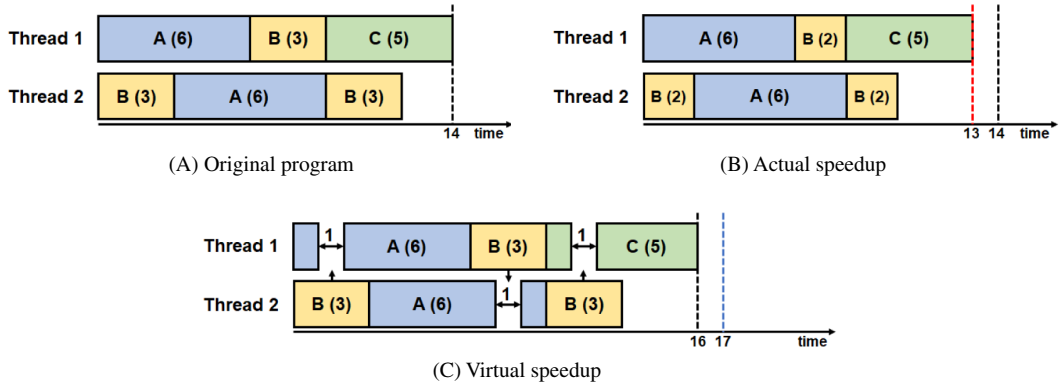

Figure 2.3: Example of virtual speedup when two threads are running and the chosen speedup line l is in function B. Every time line l in B is executed, it delays other threads by 1 time unit (tu). In this example, the original runtime is 14 tu, the sum of all inserted delays is 3 tu, runtime with virtual speedup is 16 tu, and so the potential speedup is 1 tu, which is equal to the actual speedup. Taken from [9].

Coz implements thread-level virtual speedup, which requires monitoring of the instructions each thread executes. In order to collect these data, Coz uses Linux *perf* subsystem. Using the `perf_event` API, Coz obtains both the current program counter and user-space call stack from each thread every 1ms. Ideally, the speedup requires the other threads to sleep whenever a thread executes a code line to speedup, however, this leads to considerable profiling overhead [15]. To keep this overhead low, Coz processes samples in batches of ten by default (i.e. every 10ms). Coz translates each batch to a code line and checks whether the code line is the target of virtual speedup, in which case it pauses the execution of other threads.

Evaluation. Coz was evaluated on range of highly-tuned applications: `Memcached`, `SQLite`, and the `PARSEC` benchmark suite, where it identified optimisation opportunities that have not been known. Based on Coz's causal profiles, performance of `Memcached` could be improved by 9%, `SQLite` by 25%, and six `PARSEC` applications up to 68 %. Most of the optimisations consist of modification under 10 lines of code [15].

Derived profilers. A number of related tools were proposed either to improve the causal profiling or to focus on some more specific field. SCoz tool is a system-wide causal profiler that addressed two concrete limitations of Coz: the inability to profile multiprogram applications and operating system kernel codes. Coz+ [41] is an extension of Coz for web-browser applications. TaskProf [44], TaskProf2 [45], and OMP-WhIP [12] are suite of causal profilers which implements causal profiling for identifying parallelism bottlenecks in task parallel programs. Support for causal profiling of distributed applications is implemented in DProf [11] tool.

Summary. Coz targets on finding optimisation opportunities, that may not be immediately obvious from the performance profile obtained by conventional performance profilers like *gprof*, or *perf*. Evaluation on real-life projects shows that the resulting causal profile

highlights causal relationships between hypothetical optimisations and can be more than sufficient for identifying the low-performance point of a program. On the other hand, the source line which will be virtually sped up, is randomly selected. Thus, it is questionable how the solution will work for large projects with millions of lines of code, since authors do not provide any evidence of scalability of the solution. Moreover, Coz requires access to the source code of the tested system. Despite the inevitable communication between threads, the evaluation confirm that Coz has sufficiently low overhead to be used in practice.

2.3 Performance Amplification (PerfBlower)

Performance testing framework PerfBlower [21] aims to detect memory-related issues in Java programs by amplifying the size of allocated objects. It uses domain-specific language ISL to describe the symptoms of observable performance problems on a JVM (Java virtual machine) including memory leaks, inefficiently used collections, unused return values, or loop-invariant data structures. The framework provides an automated test oracle via virtual amplification, which evaluates the presence of memory problems. Successful detection of memory-related performance is supplemented by a precise reference-path-based diagnostic information accomplished by object mirroring. The target is to blow up the effect of insignificant performance problems so that they can be captured during testing.

ISL Language. Proposed event-based language referred to as ISL (instrumentation specification language) describes symptoms of performance problems on a JVM. Since such problems cannot be expressed using logical assertions, ISL provides a pair of commands: *amplify* and *deamplify*. Command *amplify* allows to add virtual memory penalties to an object and command *deamplify* is used to remove penalties when counter evidence is observed. The symptoms, specified in ISL, are periodically checked during garbage collection.

Test oracle via virtual amplification. The amplification has per-object granularity: virtual penalties are created by maintaining a *penalty counter* (PC) inside each object. During each garbage collection, it identifies the *real heap consumption* (RHC) and then computes a *virtual heap consumption* (VHC) by adding up the PC for each object. It computes *virtual space overhead* (VSO) by comparing VHC to the RHC, which provides an automated test oracle: their experimental results show that the overall VSOs with and without real performance problems are 20+ times and 1.5 times, respectively.

Diagnostic information via object mirroring. To help developers find the root of the performance problem, PerfBlower provides diagnostic information. The so called mirror object chain of the object o reflects major reference path in the object graph, leading to object o. The chain shows both the calling context and the data structure in which a suspicious object is created [21].

Evaluation. The approach was evaluated on 13 real-life Java projects. In the evaluation, they implemented three amplifiers (ISL programs) that target memory leaks, under-utilised containers, and over-populated containers. In 5 projects they found 8 unknown problems that have not been reported in any previous work. Moreover, all problems were exposed under small workloads, thus the executions did not show any anomalous behaviour without amplification.

```
Context ArrayContext {
  sequence = "*.main,*";
  type = "Object[]";
}
Context TrackingContext {
  sequence = "*.main,*";
  path = ArrayContext;
  type = "String";
}
History UseHistory {
  type = "boolean";
  size = UP; // User Parameter
}
Partition P {
  kind = all;
  history = UseHistory;
}
TObject MyObj{
  include = TrackingContext;
  partition = P;
  instance boolean useFlag = false; // Instance Field
}
Event on_rw (Object o, Field f, Word w1, Word w2){
  o.useFlag = true;
  deamplify(o);
}
Event on_reachedOnce (Object o){
  UseHistory h = getHistory(o);
  h.update(o.useFlag);
  if(h.isFull() && !h.contains(true)){
    amplify(o);
  }
}
```

Listing 3: An ISL program from [21], amplifying memory leaks caused by unnecessary references from arrays.

Completeness. Authors also provided results of the additional experiment which examined whether PerfBlower misses bugs. This experiment includes 14 programs, which contain known previously reported performance problems. Among them, three programs could not be executed. From the remaining, all performance bugs were captured, except for one. After inspection of the source code, authors concluded that this bug could not be triggered by the given workload. In general, PerfBlower did not miss any bug that could be triggered by a test case.

Summary. The solution was evaluated on real projects and successfully amplified three types of heap data-related problems, even under small workloads. It is possible to amplify a custom heap data-related problem, but the procedure for amplifying other memory-related problems is not fully automatic, since it is necessary to design an IPL program defining the problem. The problem has to be expressed by logical statements over a history of heap state updates. Among the limitations of PerfBlower we can include that it can only find heap-related inefficiencies, the JVM needs to be rebuilt every time a new checker is added, and because PerfBlower goes along garbage collector run, its effectiveness may be affected by the garbage collecting frequency. Moreover, the authors do not provide any evidence of the scalability of the approach.

Table 2.1: The summary table of all frameworks. The aim is to compare their differences in an effort to enhance the manifestation of performance issues. PerfFuzz and Perun-fuzz belong to the group of fuzz testers, so they provide *anomalous workloads* that causes low performance. In contrast, others detect performance issues that are not workload-based. Although PerfFuzz, Perun-fuzz and Coz involves some test amplification (either the amplification by creating new tests as variants of existing ones, or amplification by modifying test execution) only PerfBlower uses so-called *virtual amplification*. There, PerfBlower does not change the test or execution of the test but only observes the test execution, performs changes, and operates on its internal data according to the given specification. An essential feature of the Coz is the computation of *virtual speed up*, which others do not pursue. By *granularity*, we mean what do framework work with as its fundamental unit, i.e. PerfFuzz counts the number of edge executions in the CFG graph, Perun-fuzz operates on the coverage of the paths in the callgraph, and Coz detects the impact of the code line and estimates the possible acceleration. Last, PerfBlower works with each object separately and perform amplification and deamplification depending on whether it meets the specified symptoms.

	PerfFuzz	Perun-fuzz	Coz	PerfBlower
anomalous workloads	✔	✔	✘	✘
virtual amplification	✘	✘	✘	✔
virtual speed up	✘	✘	✔	✘
granularity	CFG edge	CG path	line of code	object

Chapter 3

Noise Injection

Firstly, we will define the *noise injection* in the context of this work. The reader may be familiar with the term noise injection used in the context of neural networks. However, we define noise as a certain amount of additional redundant activity, whose injection into the target application should not affect its functionality but can affect its performance. For example, if we have a program that calculates a definite integral, the inserted noise should not affect the result: the result remains correct, but its computation may take longer time. Each noise injection framework differs in:

- its options of granularity for noise injection, i.e. where we can inject the noise, (e.g. before and after a routine in user-space code),
- how to create the noise, i.e. whether we can simulate the noise in the given place, or whether there are types of noise available (e.g. using `nanosleep`[1] function),
- what are the noise parameters that can be modulated, i.e. the strength of the noise (e.g. 10 nanoseconds), and the resolution of the noise parameters in case of sleep-type functions (e.g. nanoseconds, microseconds, ...),

3.1 SystemTap

SystemTap[2] is a tracing and probing framework that allows to write and reuse scripts to deeply monitor the activities of a live Linux system. SystemTap is able to extract, filter, and summarise the observed data, in order to allow the diagnosis of complex performance or functional problems [19]. In particular, SystemTap framework allows developers to write scripts for monitoring variety of kernel functions, system calls, and other events that occur in kernel space. SystemTap provides probing of kernel-space events without having to instrument, recompile, install, and reboot the kernel.

Scripting language. The core unit of the scripting language is the *probe*, which consists of a probe point (the trigger *event*) and its *handler* (the associated statements) [20]. Whenever a specified event occurs, the Linux kernel runs the handler, and then resumes. SystemTap offers several kinds of events, such as entering/exiting a function, expiration of a timer, starting or stopping the session. Handler and its statements specify what to do whenever the event happens. Handler code often includes extracting the context data, storing them in the internal variables, or printing the results [19].

[1] nanosleep — https://man7.org/linux/man-pages//man2/nanosleep.2.html
[2] SystemTap home page — https://www.sourceware.org/systemtap/

SystemTap translates the script to C and runs the system C compiler to create a binary kernel module. When the module is loaded, it activates all probed events by hooking into the kernel [19]. Example of SystemTap script can be found in Listing 4. After the session is stopped, the hooks are disconnected and the module is removed. SystemTap comes with an extensive collection of examples as well as reusable components (known as tapsets) for inclusion in user-defined scripts [37].

```
probe kernel.function("*@net/socket.c").call {
  printf ("%s -> %s\n", thread_indent(1), ppfunc())
}
probe kernel.function("*@net/socket.c").return {
  printf ("%s <- %s\n", thread_indent(-1), ppfunc())
}
```

Listing 4: Tracing and timing functions in `net/sockets.c`. The script [19] instruments each of the functions in the Linux kernel's `net/socket.c` file and prints out trace data: (1) time delta from the previous entry (in microseconds), (2) command name and the process id (PID), (3) indication of function entry ("`->`") or exit ("`<-`"), and (4) function name.

Usability for noise injection. Since Perun already uses SystemTap for its tracing collector, it could be suitable for noise injection. For a general overview of the probe point options, one can see the `stapprobes`[3] manual pages. In short, SystemTap can hang at the beginning and end of a SystemTap session, enter and return from a function within kernel space and user space, as well as a specific statement defined by either a line in the source code or an address in the application binary. In that case, we would have to have access to the source code of the application, which is not always possible, or know the addresses of the examined statements in the target binary. To simulate delays, SystemTap directly offers C-embedded functions that insert a delay into the probe handler. Another implementation option is to put an active waiting in the probe handler (active waiting can be implemented as a loop bounded in number of iterations or time elapsed), but normally such an implementation is disabled, and the compilation will fail. This is because the probe handler has to be bounded in time. The code generated by SystemTap includes checks of the total number of statements executed, which is limited (by default to 1000). A similar limit is enforced on the nesting depth of function calls as well. However, these limits can be suppressed using special SystemTap option in the so-called *guru* mode. SystemTap script that inject described noise to specified functions can be found in Listing 5.

Summary. SystemTap allows to instrument both kernel and user space code just by implementing a modest and straightforward script. Moreover, there already exists an engine for collecting the running times of target binary functions for further evaluation of noise impact. However, there is limited usage of delay/noise functions (only `mdelay`[4] to insert busy delay in milliseconds, `udelay`[5] as its microsecond twin, and noise as busy waiting) and all of them require the guru mode.

[3] stapprobes — https://linux.die.net/man/5/stapprobes
[4] mdelay — https://sourceware.org/systemtap/tapsets/API-mdelay.html
[5] udelay – https://sourceware.org/systemtap/tapsets/API-udelay.html

```
global DELAY_TIME_US = 42;
global REPS = 1000;
function tap_usleep(time_len){
  start_t = gettimeofday_us()
  udelay(time_len);
  end_t = gettimeofday_us()
  printf("%dus\n", end_t - start_t)
}
function busy_waiting(reps){
  start_t = gettimeofday_us()
  while(reps){
    end_t = gettimeofday_us()
    reps--;
  }
  end_t = gettimeofday_us()
  printf("%dus\n", end_t - start_t)
}
probe process("./a.out").function("foo").call{
    tap_usleep(DELAY_TIME_US)
}
probe process("./a.out").function("bar").call{
    busy_waiting(REPS)
}
probe process("./a.out").function("main").return{
    exit()
}
```

Listing 5: Example SystemTap script for noise injection. When target process calls `foo` function, it ensures insertion of `udelay` noise, and further calling `bar` function triggers injecting busy waiting noise, wherein while loop has to be some reasonable operation, otherwise the compiler removes the loop in order to optimise the code. Moreover, the script prints the time both types of noise take (in microseconds) and exits after catching the `main` function's return.

3.2 eBPF (extended Berkeley Packet Filter)

Let us start the description by referencing the thought of Brendan Gregg, author of the book *BPF Performance Tools* [27]: "*eBPF does to Linux what JavaScript does to HTML*". Basically, JavaScript makes a web page dynamically programmable and *eBPF* makes the Linux kernel dynamically programmable.

The original *BPF* was designed for capturing and filtering network packets, where filters are implemented as programs that run on a register-based virtual machine. Extended BPF (eBPF) is similar to original BPF, but it extends BPF by including the ability to call a fixed set of in-kernel helper functions and access shared data structures (*eBPF maps*) [2].

Running user-space code inside the virtual machine in kernel seemed like an ideal strategy for implementing observability/monitoring. Before eBPF, this was impractical since it required to change the kernel source code or load kernel modules [3]. With eBPF, user can run its own arbitrary code in the kernel without the necessity to write and build own kernel modules.

On the other hand, programming in eBPF directly is extremely difficult, so one can write eBPF programs in restricted C. In contrast to full C language, restricted C omits the use of global variables, variadic functions, floating-point numbers, passing structures as function arguments, etc. However, the development of tools that use eBPF is facilitated by BPF frontends, e.g. BCC [1] (BPF Compiler Collection) with bindings for the `Python` programming language, as shown in Listing 6.

```
from __future__ import print_function
import bcc
import time
text = """
#include <uapi/linux/ptrace.h>
BPF_HISTOGRAM(dist);
int count(struct pt_regs *ctx) {
    dist.increment(bpf_log2l(PT_REGS_RC(ctx)));
    return 0;
}
"""
b = bcc.BPF(text=text)
sym = "strlen"
b.attach_uretprobe(name="c", sym=sym, fn_name="count")
dist = b["dist"]
try:
    while True:
        time.sleep(1)
        print("%-8s\n" % time.strftime("%H:%M:%S"), end="")
        dist.print_log2_hist(sym + " return:")
        dist.clear()
except KeyboardInterrupt:
    pass
```

Listing 6: Python script for tracing of `strlen` returns. An example of using `uprobes` (user-level dynamic tracing probe) with a histogram, adopted from BCC GitHub repository [1]. After ctrl-c is pressed, it prints out the histogram of system-wide `strlen` return values.

The code is compiled into the eBPF bytecode using *Clang* compiler. The result of *Clang* compilation labelled as eBPF bytecode is standard object ELF format file that defines eBPF instructions and eBPF maps. The kernel then takes the file, and its *just-in-time (JIT)* compiler translates eBPF bytecode into native machine code for better performance. This process is depicted and described in Figure 3.1.

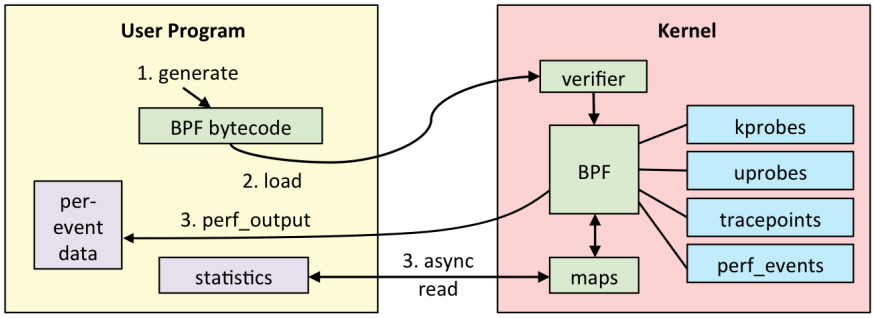

Figure 3.1: Workflow of eBPF consist of compiling the code to eBPF bytecode and sending it to the kernel using `bpf()` system call. When the program is loaded into the kernel, verifier may reject it in case it is considered as unsafe. If the BPF bytecode is accepted, it can then be attached to an event. There are different places within Linux system that can trigger BPF programs to run, which includes *kprobes*, *uprobes*, *tracepoint*, *network packets*, *perf events*, etc. Exchanging information between user-space and the kernel is provided by eBPF maps, which represents a simple *key:data* structure. Taken from [4].

Verifier. Verifier has to ensure that given eBPF program is safe for the kernel to load and run. Therefore, each eBPF program has to be safe to run until its completion. The verifier statically determines whether the given eBPF program terminates and is safe to execute [2]. The program that fail to terminate is clearly not safe, because it could be used e.g. for DoS attacks (exhaustion of computing resources).

Usability for noise injection. Although eBPF is a potent tool, implementing noise injection inside it is not possible due to its high safety requirements. The eBPF program should run for a short, limited time, and it cannot block or sleep. It is possible to import C libraries with functions forcing to sleep, but if calling these functions occurs in the program, the compilation fails. When trying to implement busy waiting as we did in SystemTap, we got a compilation error again because of hitting the maximum limit (4096) of instructions inside the program. eBPF allows only bounded loops, and user cannot suppress the checks like in SystemTap.

Summary. The eBPF framework is state-of-the-art tool for efficient monitoring, tracing, and manipulation of programs, even though it requires a fairly recent Linux kernel (at least Linux kernel version 4.4 or, preferably, 4.9). Using a BCC toolkit, it is easy to write BPF programs with front-ends in `Python` and `Lua`. eBPF is more conservative in terms of safety, because it will only run such a code that has been assumed as completely safe to run. This could be surely considered as an advantage, however, it makes implementing noise injection using eBPF impossible.

3.3 Pin

Pin is a dynamic binary instrumentation (DBI) framework that allows to create tools for dynamic code analysis of the user-space applications. The framework enables to monitor the compiled program by embedding arbitrary C/C++ code even during the program exe-

cution. Thus, it requires no recompiling of the source code and also supports instrumenting dynamically generated code.

In essence, Pin works as a *just-in-time (JIT)* compiler. However, the input to this compiler is a regular executable, not bytecode. Pin generates a new code for the basic block[6] beginning at the first instruction of the executable and then passes the control to the generated code sequence. When a branch exits the sequence, Pin ensures that it regains the control, generates more code for the branch target, and continues the execution. Pin keeps all generated code in memory so it can be reused, and direct branching from one sequence to another makes it efficient.

The tools implemented using Pin are called *Pintools*, and can be used to provide various program analysis on *Linux*, *Windows* and *macOS* user-space applications. A Pintool is a compiled binary file. For Windows systems, it is a dynamic library with a `.dll` extension, systems, for Linux it is a shared library with a `.so` extension, and dynamic library with `.dylib` extension for macOS. Technically, instrumentation inside the Pintool includes two basic elements:

- *instrumentation*, i.e. a mechanism that decides where and what code is inserted,
- *analysis*, i.e. the code to execute at insertion points.

Both components are defined in a Pintool, which is basically a plugin that modifies the code generation process inside Pin. Hence, the Pintool must share the same address space as Pin and the instrumented executable, which implies that Pintool has access to all of the executable's data, even the file descriptors and other process information [8]. Pin provides a rich, extensive application programming interface (API) for instrumentation at different abstraction levels [34]:

- *image:* access to an entire image (i.e. analysed binary executable including loaded shared libraries), where Pin can iterate over program sections, routines in a section, and instructions in a routine;
- *routine:* access to one routine and iteration over instructions of the routine;
- *trace:* instrumentation of one trace at a time by starting from the current instruction and ending with an unconditional branch (it involves also routine calls and returns);
- *instruction:* lowest level of instrumentation, processing one instruction at a time.

JIT & Probe Mode. Pin supports two modes of instrumentation: JIT and Probe. In JIT mode, Pin creates a modified copy of the application on-the-fly, and the only code that is executed is this generated code. The original code is used only for reference and never executed. When generating the code, Pin allows the user to inject instrumentation code. JIT mode utilises a just-in-time compiler to recompile the original code and insert the instrumentation. On the other side, Probe mode instrumentation is based on the *code trampoline* technique. In Probe mode, Pin modifies the original code with *probes*, where probe is a jump instruction that overwrites instruction(s) in the application and invokes the instrumentation. Since the application and analysis routine run natively, it improves the performance and (dramatically) reduces overhead. However, in Probe mode, we can only instrument on a routine level, i.e. probes can be placed only at the routine boundaries.

[6] a sequence of instructions terminated at a control-flow changing instruction (single entry, single exit)

When inserting probes, Pin copies and translates original bytes, thus replaced functions can be called from replacement function, as can be seen in the Figure 3.2. Whilst Probe mode can operate with a limited subset of all Pin features, JIT mode supports all Pin's features, making this approach more flexible and common, in contrast to less flexible but low-overhead Probe mode [16].

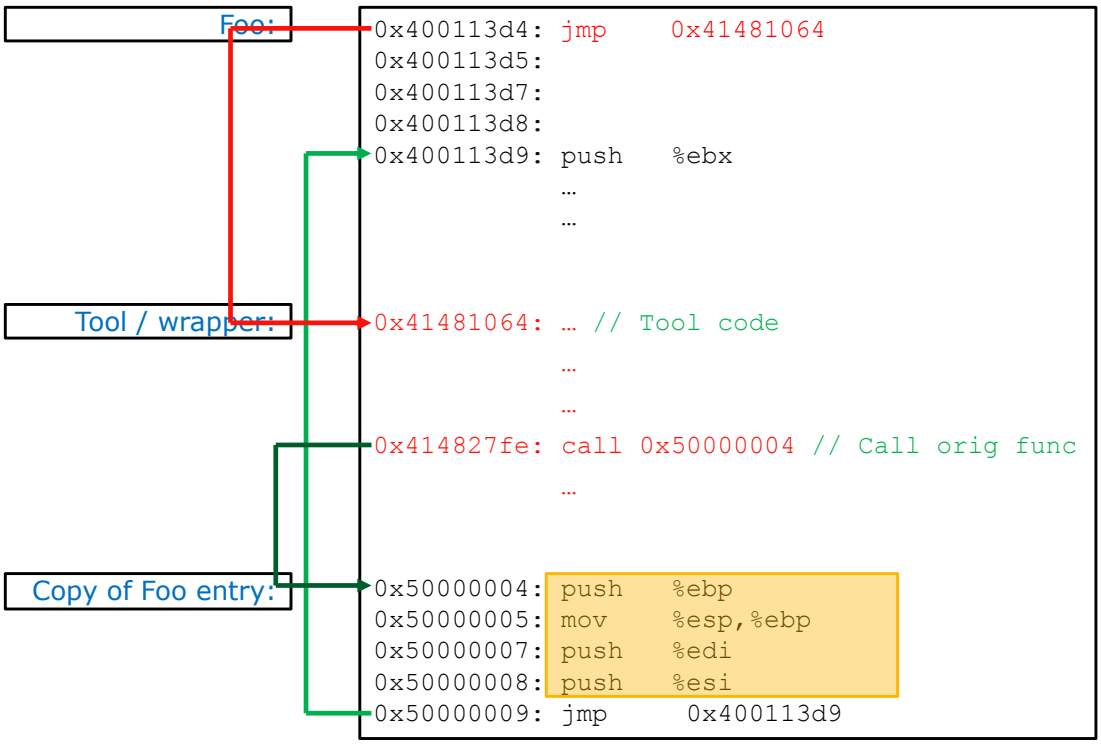

Figure 3.2: A sample probe on function `Foo`, from [16]. In this example, Pin copies and translates original bytes of function `Foo` entry from address range (`0x400113d4`, `0x400113d8`) to (`0x50000004`, `0x50000008`) and right after the copied bytes it inserts unconditional jump instruction back to the function continuing (to address `0x400113d9`). Instead of the start of the original `Foo` entry (at address `0x400113d4`) is now probe, so a jump instruction to the Pintool code, whence the replaced function can be called.

Usability for noise injection. Pin has been successfully used in [22] for random noise injection into the run of a concurrent program to stimulate rare (but legal) interleavings that may yield so far undiscovered errors. Thanks to the wide levels of granularity that Pin can work with, it is suitable for noise injection for the purposes of various analyses. The Pin API even contains a rich suite of delay simulation, which includes thread sleeping or forcing the thread to yield the processor. Example of Pintool that injects sleep noise can be seen in Figure 7. Moreover, Pin does not contain any complexity controls of instrumentation callback, which offer the possibility of implementing a delay even by busy waiting [22].

```
#include "pin.H"
#include <fstream>
#define SLEEP_TIME_MS 42
#define FUNCTION "foo"

using namespace std;
ofstream OutputFile;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
                            "output", "specify output file name");

VOID Analysis(){
    PIN_Sleep(SLEEP_TIME_MS);
    OutputFile << PIN_GetTid() << " was sleeping for "
               << SLEEP_TIME_MS << " ms." << endl
               << flush;
}

VOID Instrumentation(IMG img, VOID *v){
    RTN rtn = RTN_FindByName(img, FUNCTION);
    if (RTN_Valid(rtn)){
        RTN_Open(rtn);
        RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)Analysis, IARG_END);
        RTN_Close(rtn);
    }
}

int main(int argc, char *argv[]){
    if (PIN_Init(argc, argv)) return EXIT_FAILURE;
    PIN_InitSymbols();
    OutputFile.open(KnobOutputFile.Value().c_str());
    IMG_AddInstrumentFunction(Instrumentation, NULL);
    PIN_StartProgram();
    return EXIT_SUCCESS;
}
```

Listing 7: Source code of Pintool working on image level instrumentation in JIT mode, injecting noise in the form of calling `PIN_Sleep` function. The Pintool writes the information about what thread was forced to sleep (i.e. called function `foo`) to the output file `output`.

Summary. Pin offers easy-to-use dynamic instrumentation, with rich API for instrumentation and analysis. Pin is capable of instrumenting multi-process and multi-threaded applications, however, it support instrumenting only user-space code. Various types of noise can be injected on a different instrumentation levels, on real-life applications, like databases or web browsers [16]. Though, it is necessary to implement side runtime data collector for evaluating the noise effects, with the existing limitation of time data obtaining in Pin (see Section 5.2.2). Nevertheless, Pin is multiplatform instrumentation tool, that can be possible to used in Perun for further analyses in the future.

Table 3.1: Summary table of frameworks with respect to noise injection requirements defined at the beginning of this chapter. *Granularity* property describes the framework's ability to inject the noise at different instrumentation levels. SystemTap and eBPF allow to instrument concrete instruction or line of code; however, this requires knowing the instruction address, resp. the address corresponding to the selected line of code. Since we encountered that eBPF disallowed noise injecting in our manner, we denote no *noise options* for eBPF, but SystemTap and Pin offer several options for noise simulating. With noise options are directly linked their *parameters*, which are expressed by different resolution in case of sleep-based noise, by the number of iterations in busy waiting loop and by the number of times a thread yields the processor in case of `PIN_Yield`.

	SystemTap	eBPF	Pin
granularity	routine, line of code*, instruction*, (user/kernel space)	routine, line of code*, instruction*, (user/kernel space)	image, routine, trace, instruction (user space)
noise options	`mdelay`, `udelay`, busy waiting	none	`PIN_Sleep`, `usleep`, `nanosleep`, `PIN_Yield`, busy waiting
parameters	milliseconds, microseconds. iterations	none	milliseconds, microseconds, nanoseconds, yield count, iterations

Chapter 4

Analysis of Requirements

In this chapter, we will specify the functionality that the resulting analyser should implement. The main aim of this specification is to outline the purpose, the capabilities and the limitations of the developed project. In particular, we will start with short overview, followed by list of *functional* and *non-functional* requirements.

4.1 Overall Description

The goal of this work is to build a framework that will help developers detect performance bottlenecks or optimisation opportunities. The work is mainly based on the *perfblowing* technique: artificial inflating of program performance based on noise injection. Using this noise, the one then observes its effect on overall program performance (e.g. on the program runtime). Depending on the goal of perfblowing, we can distinguish two modes of the framework: *slow-down* and *speed-up*.

The noise will be injected by dynamic instrumentation framework Pin, with the following *parameters*: noise *location*, noise *type*, and noise *strength* (e.g. we inject 1 millisecond (strength) of `PIN_Sleep` (type)to function `foo`). Considering Pin's capabilities, we could inject the noise before or after any SUT binary instruction, including loaded libraries, thus on the lowest level possible. However, such an approach can be inefficient, whereas the SUT can consist of a huge number of instructions (we have already discussed Pin capabilities in Section 3.3). Hence, we decided to focus on injecting noise in functions only.

Blindly injecting noise into random functions is straightforward, but most likely ineffective. Therefore, we propose to adapt the concept of fuzzing to repeatedly run *perfblowing experiments* whilst collecting the runtime data of SUT.

We call the performance experiment p the tuple (f, t, s, m), such that if: (1) $m =$ *slow-down* the noise is injected at function f of noise type t and with strength s, and (2) if the $m =$ *speed-up* the noise is injected in all other functions except the f. The gathered runtime data will be used to refine future perfblowing experiments, e.g. modifying the noise type and its strength.

After the perfblowing process, the developer will see the perfblowing outputs, including all logging messages printed during the process, readable raw text results and visualised causal profiles, all according to chosen perfblowing mode and launching options.

4.2 Functional Requirements

The resulting Perun-Blower will provide the following functionalities:

(1) **FR_PERUN (Implemented in Perun):** the perfblowing framework must be integrated into the Perun framework.

(2) **FR_BTC (Built upon Tracer collector):** the perfblowing loop will be built upon the infrastructure of Tracer, and will extend its existing suite of profiling engines.

(3) **FR_COMP (Compatibility with existing trace engines):** the Tracer's Pin engine developed in this work must be compatible with other Tracer engines.

(4) **FR_INIT (Initial corpus selection):** the set of candidate functions for perfblowing must be pre-selected by one or more selection methods (e.g. most called functions, functions with most call-sites, input/output functions) described in Section 6.1.1.

(5) **FR_CSF (Custom set of functions):** the framework has to provide an option for a user to specify a custom set of functions, which will be subjected to perfblowing.

(6) **FR_FS (Selection of a function):** before every iteration of the perfblowing loop, one function has to be selected for noise injecting based on distinct strategy.

(7) **FR_INOF (Inserting a noise into one function):** the framework must be able to inject a noise into one candidate function (the so-called *slow-down* mode).

(8) **FR_INAF (Inserting a noise to all but one function):** the framework must be able to inject a noise to all but one candidate function (the so-called *speed-up* mode).

(9) **FR_EVAL (Impact evaluation of the injected noise):** depending on the mode, the framework must automatically evaluate how injected noise impacts the SUT performance and change the perfblowing state accordingly.

(10) **FR_RATE (Functions rating as a candidate for noise injection):** every function in the corpus must have assigned numeric value, that represents its quality as a perfblowing candidate.

(11) **FR_PAR (Evaluation-based noise parameter refinement):** the noise parameters must be adapted in regard to previously performed evaluations.

(12) **FR_LOG (Log information about perfblowing state during run):** the information about perfblowing state must be regularly updated and presented to the user.

(13) **FR_CTRL (Controlling the runtime of perfblowing):** perfblowing loop has to be bounded by specified number of perfblowing experiments or specific time, so the user can control how long will the perfblowing loop run.

(14) **FR_VIR (Readable visualisation and interpretation of perfblowing results):** the framework has to present achieved results in a readable text and graphical form.

4.3 Non-functional Requirements

While functional requirements specify what the framework should, non-functional requirements describe other aspects of its development. Below we propose a list of non-functional requirements this work should follow:

1. **NFR_SCA (Scalability):** the framework architecture have to be designed and implemented so that perfblowing experiments will scale reasonably well.
2. **NFR_MAIN (Maintability):** the implementation should be generic enough to be simply extended or modified in a possible future work (either as a whole project or as a separate components).
3. **NFR_REL (Reliability):** the framework has to be adequately tested with the automated test suite to ensure a high level of reliability.
4. **NFR_NIO (Minimised overhead of noise injection):** the Pintool used as the noise injector in the perfblowing should have low overhead.
5. **NFR_DCO (Minimised overhead of time data collection):** the Tracer Pintool used for collecting data about SUT time stamps should have low overhead.
6. **NFR_PCHO (Low overhead of code generation and injection):** the adjusting of noise parameters in each perfblowing loop should involve the least possible work.
7. **NFR_DCA (Reasonable accuracy of data collection):** the SUT runtime data obtained by Tracer tool has to have reasonable accuracy without skewing the collecting code effectiveness. The requirement aims at the choice of method used for data collecting.
8. **NFR_DEP (Minimised number of dependencies):** the work should have as few dependencies as possible (Python packages or another required software).
9. **NFR_QUA (Quality of use):** the perfblowing should be performed on SUT with a small amount of prerequisite work, thus if the framework can automate some actions, it should not demand this work from the user.

Chapter 5

Extending Tracer with Pin Engine

The current version of Perun (0.20.2) uses the Tracer collector as its main profiler. Tracer currently supports only SystemTap and eBPF instrumentation frameworks. In Chapter 3, we have shown, however, that both frameworks do not sufficiently support noise injection. Hence, in this chapter we show how to extend the Tracer's engines with Pintool-based engine to allow instrumentation using the Pin framework. First, we will briefly list the program interface of Tracer: the so-called *engines*. Then, we will show how to implement the individual functions based on the Pin framework.

5.1 Tracer

Tracer (or *Trace collector*) collects profiling data about running times for each of the specified functions or USDT (User Statically-Defined Tracing) probes in the code. Design of Tracer architecture allows supporting multiple underlying instrumentation frameworks (such as SystemTap or eBPF) using the so-called *engines*. Each engine must implement the so-called *Common Engines Interface* that unifies communication with concrete engines. The interface is defined as the set of functions (where →represents the returned type) [38]:

- `check_dependencies:` checks that all of the engine requirements are satisfied and all of the dependencies are available.
- `available_usdt` → `dict`: extracts the USDT probes from the SUT.
- `assemble_collect_program:` assembles the collection program according to the specification from the user.
- `engine_collect:` runs the data collection program.
- `transform` → `generator:` transforms the raw output of the individual engine to the unified Perun resources.
- `cleanup:` frees the set of resources that have to be cleaned up.

5.2 Pin Engine

We will now provide an overview of the newly implemented Tracer engine based on Pin. This engine will collect functions' runtime in individual threads and processes within the user-

space of the target binary. The engine uses the triple of Pin, Tracer Pintool[1] and SUT to run the profiling. Tracer Pintool (detailed in Section 5.2.1) is used as the core of the Pin Tracer Engine. It contains all the instrumentation code needed for collecting the runtime data of the specified functions.

When invoking the `perun collect` command (i.e. the main command that profiles the given SUT), the Tracer calls the selected set of functions implemented within the chosen engine. In the Pin engine, we first check whether the path to the Pin binary is valid inside the `check_dependencies` function; the user is notified in case of missing the Pin framework. Since Pin does not support working with the USDT probes, we cannot extract them from the SUT either. Thus the `available_usdt` function returns the empty Python dictionary. Next, we generate the C++ code of the Tracer Pintool and build it with our Makefile scheme (described in Section 5.4) inside the `assemble_collect_program`. When the Tracer Pintool is successfully built, we execute it with the Pin and SUT and collect data by running the external command in the following form: `pin -t tracer_pintool.so -- SUT_binary [args] [workload]`, where `args` and *workload* represent the SUT arguments and workload, respectively. This command is run in the `engine_collect` function. The result is an output file with data in raw format, specified in Section 5.2.3. We chose the same format as is used in the SystemTap Tracer engine, and we can reuse the `transform` function calls from the SystemTap engine for transforming the collected data into the Perun profile resources. The last step is to perform the `cleanup`, which in our case deletes the output file(s) created during collecting process. No other resources are required.

5.2.1 Tracer Pintool Overview

In this section, we will outline the concept of the Pintool that represents the core of Pin Tracer engine and that implements the `assemble_collect_program` and `engine_collect` functions. Pin instrumentation can work in two modes: *JIT* and *Probe* (we described these modes in Section 3.3). Their main difference (in the context of Tracer) is that in Probe mode, we can only instrument routines (at the image level, i.e. instrumentation is performed when loading an image[2]). In contrast, in the JIT mode, it is possible to use all levels of instrumentation, but with more significant overhead compared to the Probe mode. We decided to implement trace Pintool in both modes, although we recommend measuring with the Probe mode, as the measurements are more accurate due its lower overhead. Below, we outline the implementation details of trace Pintool in both modes.

In order to focus on the functions that are reachable with the SUT only, we restricted ourselves to measure the time elapsed in function calls reachable from the `main` function of the binary executable. Moreover, we allow the user to define different `main` function so the scope of measuring can be customised. However, further we will assume that the scope of `main` function is the scope we are interested in. Hence, we register callbacks that set the global flag during the run of SUT, denoting whether we are or are not in the `main` scope.

5.2.2 Collecting Performance Data

The result of the perfblowing should be a performance and causal profile, both based on time data of function calls. Therefore, in out Pintool we focus our interest on collecting

[1] We repeat that Pintools are basically tools implemented using Pin.

[2] In Pin, analysed binary executable including loaded shared libraries are represented as images.

function runtimes for each call in SUT. In addition, we utilise these times when we want to analyse whether the noise injection had an impact mainly on the total runtime of SUT and to diagnose if we discovered a bottleneck or an opportunity for optimisation.

General trace engine idea

In order to obtain runtime of individual functions within our Pintool, we use a straightforward approach: we get the timestamps before and after the function call and subtract these values to get the runtime. The approach is illustrated with pseudocode in Listing 8. Identical idea is used in Tracer engines that are already implemented within Perun. However, they use SystemTap and eBPF for instrumenting the code. The tricky part is, however, choosing the most precise handler of working with timestamps and clocks.

```
1 time begin = get_current_time()  // instrumented code
2 call f()
3 time end = get_current_time()    // instrumented code
4 // execution time = end - begin
```

Listing 8: Pseudocode of obtaining execution time of function `f`.

Measuring runtime with Pin

This section analyses the possibilities of measuring time using Pin and discuss the choice of the function that will actually implement the function `get_current_time()` from Listing 8. Generally, Pin allows us to insert code written in C/C++, so below we describe several options we considered to acquire current time in C/C++ inside a Pintool:

(1) calling `time`[3]: returns the time from the OS since the Epoch (the Unix epoch is the time 00:00:00 UTC on 1 January 1970), however, its resolution is in seconds. Hence, it cannot be used to precisely measure runtimes in milliseconds, which to our experience are quite common.

(2) calling `clock_gettime`[4]: one of the recommended functions for obtaining the exact time with the possibility of selecting the exact clock (e.g. `CLOCK_MONOTONIC`), with nanosecond resolution; however, Pin uses its own `libc` replacement, which does not implement all functions of the standard `libc`, which is why launching the Pintool with calling `clock_gettime` function raises runtime error.

(3) calling `getrusage`[5]: returns resource usage for calling process, calling process children or calling thread with microsecond resolution; usage within a Pintool is not possible due the same reason as with `clock_gettime`, the execution fails with runtime error.

(4) calling `gettimeofday`[6]: similarly to `time`, it returns the time since the Epoch, but in structure with seconds and microseconds values separately and according to the specified timezone. The two-item structure brings the drawback in the situation of com-

[3] time — https://linux.die.net/man/2/time
[4] clock_gettime — https://linux.die.net/man/2/clock_gettime
[5] getrusage — https://linux.die.net/man/2/getrusage
[6] gettimeofday — https://linux.die.net/man/2/gettimeofday

puting the runtime as the difference of two values, where we have to convert the both resulting structures to pure microsecond values and then subtract them.

(5) using `std::chrono`[7] library: available in `C++11`, however, a Pintool cannot be compiled with this library and ends with error.

(6) calling `OS_Time`[8]: function is included in the `PinCRT` library (Pin's set of functions that provides a generic way to interact with the OS) and retrieves current time in microseconds since the Epoch. In contrast to `gettimeofday`, which returns a structure with the items containing seconds and microseconds, `OS_Time` returns the result as an unsigned 64-bit integer directly in microseconds, thus no additional computation has to be performed when trying to calculate the difference between two time values (e.g. before and after the function call). In addition, custom experiments showed that `OS_Time` consumes less or equal computation time than `gettimeofday`.

(7) using `IARG_TSC` value: Pin offers to pass the *Time Stamp Counter* (TSC) value at the point of entering the analysis call; however, TSC clock frequency actually varies depending on the hardware and may vary during runtime.

As a best solution for acquiring the current time was generally recommended `PinCRT` library function `OS_Time` which gives the current time since Epoch in microseconds. The summary of the described time obtaining approaches can be found in Table 5.1.

Table 5.1: Summary table comparing various time obtaining methods within a Pintool. Column *direct value* denotes whether an approach returns the time value in such format that the difference between two values can be computed by subtraction, without additional post-computing. We consider an approach *applicable* if its usage does not trigger any error, has at least microsecond resolution and retrieves real-time value, not just the number of cycles elapsed like TSC.

	resolution	direct value	applicable
`time`	seconds	✔	✘
`clock_gettime`	nanoseconds	✘	✘
`getrusage`	microseconds	✘	✘
`gettimeofday`	microseconds	✘	✔
`chrono`	nanoseconds	✔	✘
`OS_Time`	microseconds	✔	✔
`IARG_TSC`	-	✘	✘

5.2.3 Tracer Pintool Output

When the `engine_collect` function successfully profile the SUT, Pin Tracer engine produces the output file with the collected data. We record each collected information in the output file as soon as we obtain it. To be compatible with existing Tracer engines, the output file is in so-called *raw* format based on the four types of records: `THREAD_BEGIN`, `THREAD_END`, `FUNC_BEGIN`, `FUNC_END`. THe output is then processed using the `transform` method. The format of the records is shown in Listing 9 below.

[7] std::chrono library — https://en.cppreference.com/w/cpp/chrono

[8] OS_Time — https://software.intel.com/sites/landingpage/pintool/docs/97503/PinCRT/html/group__OS__APIS__TIME.html

```
// thread-specific record
{THREAD_BEGIN_ID | THREAD_END_ID} tid pid ppid timestamp;SUT_Name
// function-specific record
{FUNC_BEGIN_ID | FUNC_END_ID} tid timestamp;Function_Name
```

Listing 9: Format of the raw output file from Pin Tracer engine. Each line contains one record related to some thread or function boundary event (begin or end).

In the case of thread-specific records, there is a difference in obtaining the timestamp in Probe and JIT mode. While in JIT we are allowed to use callbacks to handle these thread events[9]; Probe mode does not support it. Thus, we record the `THREAD_BEGIN` of the thread x when the first function call within thread x is observed. Symmetrically we work with the record `THREAD_END`. The record carries the timestamp of the last function call within the thread. However, during the collecting, we do not know if the currently processed function return is the last within the thread; therefore, we need to record all of them. These records are written to a separate output file, which is processed after the collection, and the `THREAD_END` records are added to the main output file.

The raw format also allows other types of records (e.g. `PROCESS_BEGIN` and `PROCESS_END` for handling the process events or `USDT_BEGIN`, `USDT_END`, and `USDT_SINGLE` specific for USDT probes). However, Pin does not support acquiring these. An example of raw output file is in Listing 10.

```
5 16709 16709 16369 1621677683676663;APP
0 16709 1621677683892141;bar
1 16709 1621677683892165;bar
5 16712 16712 16709 1621677683903436;APP
0 16709 1621677683905363;foo
5 16713 16712 16709 1621677683920351;APP
0 16713 1621677683923642;foo
1 16709 1621677683935433;foo
1 16713 1621677683962950;foo
6 16713 16712 16709 1621677683972490;APP
0 16712 1621677683986758;foo
1 16712 1621677684013197;foo
0 16712 1621677684015543;foo
1 16712 1621677684042267;foo
6 16712 16712 16709 1621677684072705;APP
6 16709 16709 16369 1621677684099036;APP
```

Listing 10: An example of raw output file, with record types: `THREAD_BEGIN(5)`, `THREAD_END(6)`, `FUNC_BEGIN(0)`, and `FUNC_END(1)`. Output contains runtime data of two functions (`foo`, `bar`) within the three threads in total (`16709, 16712, 16713`).

[9] by registering the callbacks with `PIN_AddThreadStartFunction` and `PIN_AddThreadFiniFunction`

5.3 Designing a Pintool

In this work, we will use several Pintools, each with a different aim: (1) Tracer Pintool that collects runtime data of the functions, (2) Pintool for noise injection, used for our perf-blowing approach and (3) function selection Pintool that analyses the given SUT function calls (serves as preparation for noise injecting). In the following sections, we will outline the key components of these Pintools, their configuration, and the building process. Pin and Pintools can be used for multiple dynamic analysis purposes. In general, Pintools may include code for one or more analyses (usually when these analyses do not affect each other; when one analysis requires results from the other, they must be divided into separate Pintools). Pintool then runs the Pin and the target binary for analysis. We use Pin for several different dynamic analyses and one of them is the time data collection of binary's functions, in order to satisfy **FR_BTC (Built upon Tracer collector)** requirement.

As we mentioned in Section 3.3, Pintool code mainly consist of *instrumentation* and *analysis*. However, Pintool also includes *notification callbacks* which are invoked when an event like thread creation or forking occurs. These callbacks are generally used to gather data, initialisation or clean-up [8]. The notification callbacks as well as instrumentation callbacks must be registered within Pintool main function before running the target binary.

First we designed an universal interface for generating Pintool in Perun, which can be extended for any dynamic binary analysis. We divide the code of generic Pintool into several basic blocks (which together forms the so-called *Pintool skeleton*) in order to make the code more understandable and maintainable. The skeleton of the general Pintool then consists of the following blocks:

- `includes`: code with included libraries and header files,
- `globals`: declaration and definition of global variables,
- `helpers`: inlined helper functions,
- `analysis`: code of analysis routines,
- `instrumentation`: instrumentation callbacks,
- `parallelism management`: notification callbacks around forks, thread starts and ends,
- `fini`: callback for `fini` function, which is called when the target application exits,
- `main`: the main function of the Pintool, which generally consist of initialisation, opening of the output files, registration of the callbacks and executing the target application.

In addition to Pintool itself, there is also a configuration header `config.h` which can be used to group all optional Pintool configuration settings. For example, we use the C Preprocessor directive `define` to define the constant `PROBE`, and together with conditional compiling, we can compile the code of Pintool for either Probe or JIT mode just by setting the value of this constant. It is also an appropriate place to define Pintool-specific global variables. The whole scheme of building Pintool in Perun is shown and described in the following section.

5.4 Pin's Makefile Infrastructure

In general, Pintools are built using `gnu make`[10] on all target platforms. The directory `source/tools/Config` inside the Pin holds the common make configuration files, which may serve as a basis for user's makefile. In our scenario, we use as a template two files:

- `makefile.config`: the first file to be included in the make include chain, holds documentation of all the relevant flags and variables available to users; and
- `makefile.default.rules`: the default make targets, test recipes and build rules.

We use the modified copies of these listed files in order to improve the build process of the Pintool and to allow the built of the Pintool in case that its code is divided into more files (e.g. `config.h` in our case). The scheme of build and run of a Pintool is depicted in Figure 5.1. Below the scheme, we list the concrete changes that had been made in order to compile Pintools according to this scheme.

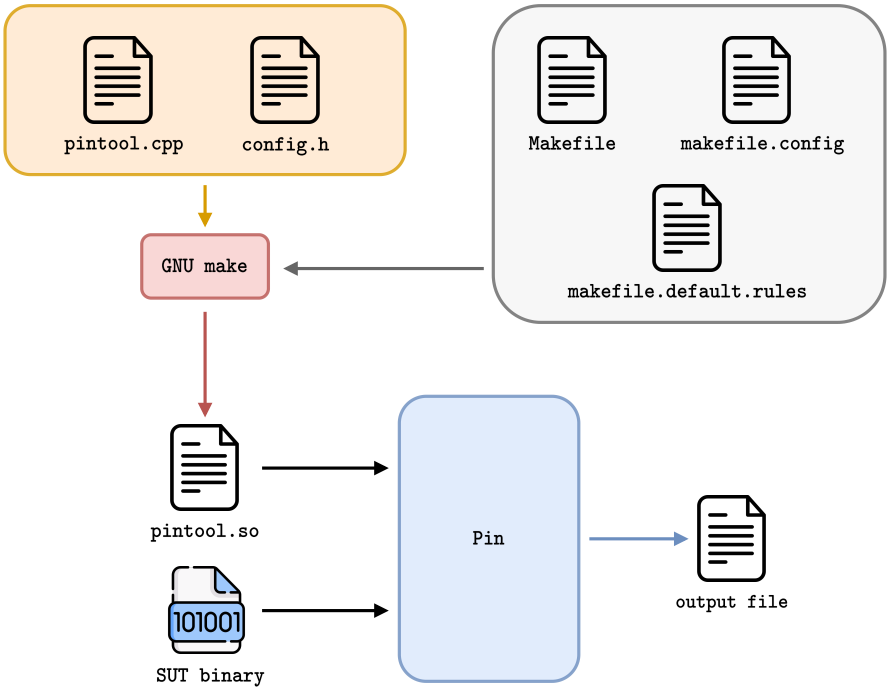

Figure 5.1: The scheme of building the Pintool and running it with Pin. The file `pintool.cpp` contains the source code of the Pintool, i.e. the Pintool skeleton. The additional information for the Pintool should be included within `config.h` header file. `Makefile`, is a general makefile, suitable for all Pintools in Perun, invoked when running `make`. By running `make` we obtain `pintool.so` library (in Linux) which can be then passed to Pin along with SUT binary. The output of running Pin can one or more output files (depending on the implementation of Pintool) with information collected during the run.

Changing default make target in `makefile.config`. By default, with unmodified config, the make builds not only the Pintool but also tries to make other targets (e.g. `objects`, `libs`, `dlls`). These builds are not necessary in our case, so skipping them reduces the overhead of building the Pintool.

[10] gnu make — https://www.gnu.org/software/make/

Adding `config.h` as a prerequisite to the Pintool build rule. Pintool's behaviour must depend on its current configuration in the `config.h` file, so its compilation must be based on the current version of this file. By adding a prerequisite to its build rule, Pintool has to be re-builded every time this file is modified too.

Chapter 6

Design and Implementation

In the Chapter 4, we outlined the overall idea behind the Perun-Blower. Here, we will elaborate the details of the tool design, and show how we will tackle particular issues. The main loop is depicted in the pseudocode in Listing 14. In the following, we discuss the individual parts of the algorithm:

(1) **initial function selection:** how to analyse SUT functions in order to define the set of functions that will be measured and used as a corpus (i.e. initial set of functions for perfblowing experiments),

(2) **obtaining baseline runtime data:** how to utilise the Pin Tracer engine (in particular, implementation of methods: `assemble_collect_program` and `engine_collect`) to collect the baseline runtime data of the SUT functions

(3) **corpus refinement:** how to enrich of the corpus with other functions, based on the measured runtime data,

(4) **building the Pintool for noise injection:** how to generate the code of the Pintool for noise injection and build it,

(5) **selecting a candidate:** how to select the function from the corpus as a subject for the perfblowing experiment,

(6) **defining the remaining noise parameters:** how to determine the used noise type and its noise strength,

(7) **running the perfblowing experiment:** how to run the SUT with the injected noise

(8) **handling the perfblowing experiment outcome:** how to process the output data from the perfblowing experiment,

(9) **results interpretation:** how to present the results to a user as text or graphical interpretation.

Steps (1)-(4) forms the *initialisation phase*, which we describe in Section 6.1. The *perfblowing loop* then includes the iterative repeating of the steps from (5)-(8). Section 6.2 contains the detailed description of these steps. Last step (8) is described in Section 6.3, together with the examples of the outputs.

6.1 The Initialisation Phase of Perfblowing Experiments

Perfblowing algorithm starts with the so-called initialisation phase that serves as a necessary preparation before the actual perfblowing loop. This phase consists of four steps, which we will explain in the following sections, outlined by the pseudocode in Listing 13.

6.1.1 Initial Function Selection

First, we create a corpus, i.e. set of SUT's functions to which we will inject the noise during single iteration of the perfblowing loop. The selection is performed by running the Pintool (so-called `FunctionSelector`), which analyses all the function calls during the SUT run with user defined arguments and workload, and returns two kinds of results:

- **functions to measure**: list of functions called from the SUT `main` function; or the user can define its own `main` replacement using the `--main` option, and
- **functions' calls statistics**: data about function calls and call-sites[1] during the SUT run with user specified arguments and workload, from which we derive number of functions' calls and call-sites (for more information see Function Selector Pintool description below)

To form the corpus, we parse the output file of the `FunctionSelector` Pintool and select the functions that met the predefined criteria. Function f is added to the corpus if f meets at least one of the criteria:

(1) $called(f) >= threshold_{called}$: function f is called at least $threshold_{called}$ times (the threshold can by specified using the `-called-threshold` option; the value is set to 2 by default, which we assume is a mild condition for addition to corpus);

(2) $call\_sites(f) >= threshold_{call\_sites}$: function f is called from at least $threshold_{call\_sites}$ call sites (value of the threshold can by specified using the `-call-sites-threshold`, the value is again set to 2 by default);

(3) $complexity(f) > O(1)$: the estimated time complexity of function f, inferred by the Perun `bounds` collector, is higher than constant complexity;

(4) $io(f)$: function f is an input/output function (we consider the function to be input/output if its name has prefix `'_IO'`); note that this criterion is optional and is disabled by default and it can be enabled by using `--io-functions` option,

(5) $locking(f)$: function f includes locking functions (here we limit ourselves to the set of `pthread` functions[2]); this criteria is also disabled by default and can be enabled by using `--locks` option.

We proposed these criteria to focus on functions that are more likely to affect the performance. Forming the corpus based on the criteria listed above is suitable for slow-down mode of perfblowing, i.e. finding the bottlenecks. In case of speed-up mode, we have to primarily take into account the average runtime of the functions, which we obtain later in the initialisation phase (see Section 6.1.3).

[1] locations where the function is called

[2] `pthread_mutex_lock`, `pthread_mutex_trylock`, and `pthread_mutex_unlock`

User can also use our perfblowing framework with `--show-functions` option to just print the `all_f` and `corpus` sets of functions. Moreover, we use `c++filt`[3] to demangle (decode) function names (in case that compiler used *name mangling*[4]), so all the functions are interpreted to user in both mangled and demangled form. An example of the framework output with using `--show-functions` option is shown in Listing 11. On the other hand, user can define custom corpus by specifying the path to the file, where the functions of corpus should be listed (one row for each function name). However, since Pin works with the function name in the mangled form, the function names in the corpus file has to be in mangled form.

```
ALL FUNCTIONS (204):
  _ZN3re211FilteredRE2C1Ev <==> re2::FilteredRE2::FilteredRE2()
  _Znwm <==> operator new(unsigned long)
  __tls_get_addr <==> __tls_get_addr
  _ZN3re26Regexp7DestroyEv <==> re2::Regexp::Destroy()
  _ZN3re24ProgC2Ev <==> re2::Prog::Prog()
  ...
CORPUS (154):
  _Znwm <==> operator new(unsigned long)
  _ZN3re26Regexp7DestroyEv <==> re2::Regexp::Destroy()
  ...
```

Listing 11: Example output of using `--show-functions` option.

Function Selector Pintool. The code of this Pintool is generated every time a perfblowing is launched to analyse the SUT binary and provides the information about the function calls. This Pintool can work in both Pin modes: JIT and Probe, with JIT mode set by default. This is because, unlike in the Probe mode, where when registering instrumentation callbacks we have to iterate over all functions in each loaded image and instrument them, in JIT mode, we utilise the `Routine` level instrumentation instead. Thus the instrumentation is performed dynamically whenever a new function is called. In all other Pintools used in the perfblowing, we prefer to use the Probe mode since there we can count on the already specified list of all functions; therefore, the iteration over all functions in each image is not necessary. However, we allow the user to switch to the Probe mode for `FunctionSelector` Pintool with the `--probe-fs` option. The `FuntionSelector` Pintool has two output files: (1) file with the all called functions in the `main` scope and (2) file with the records of all functions' calls in the `main` scope with the return address for each function call (see `IARG_RETURN_IP`[5] instrumentation argument), so we can then derive how many calls and call-sites does the function have. The final result of this analysis is: (1) list of the functions called in the `main` scope (`all_f`), and (2) the `corpus`, i.e. the list of the functions that meets the specified criteria.

[3] c++filt — https://linux.die.net/man/1/c++filt

[4] name mangling — https://en.wikipedia.org/wiki/Name_mangling

[5] Pin instrumentation arguments — https://software.intel.com/sites/landingpage/pintool/docs/98314/Pin/html/group__INST__ARGS.html

6.1.2 Obtaining Baseline Runtime Data

After every perfblowing experiment we need to compare the runtime data with the noise injected to some baseline data, to evaluate the effect of injecting the noise. Hence, we need to obtain them in the initialisation phase. In our case, the baseline data are derived from the functions' runtimes data. For this purpose, we need to generate the code of the `Tracer Pintool`, build it and run with the specified list of functions to measure (`all_f`).

We already described the Tracer Pintool output file and its format in Section 5.2.3. The baseline data are based on this output file and contains the following this types of information:

- *f_times*: the sum of all function calls' runtimes for each measured function;
- *exec_data*: data for each function execution (tuple `begin`, `end`) divided by the thread id (tid) of the execution (necessary for the visualisation showed in Section 6.3.2);
- *total_runtime*: number of microseconds elapsed from the first to the last recorded timestamp;
- *f_calls*: total number of calls of each function;
- *min_begin*: the value of the lowest timestamp recorded.

The speed-up mode of the perfblowing needs the following additional information, which will be used for estimation of the potential speed-up.

- $f\_calls\_in(t)$: number of calls of f in thread t (identification by tid);
- $not\_f\_calls\_in(t)$: number of calls of other functions than f in thread t (identification by tid);
- *t_times*: the total runtime of the threads (identification by tid);

However, in case of the data stored in *t_times*, *f_calls_in_t*, and *not_f_calls_in_t* structures, we cannot identify the information only by the tid, because in the each individual run, the tids will be different. Thus, we would not be able to compare two runs, since the threads would not be numbered equally. Hence, we rename the tids w.r.t. definition order. An example of renaming is shown in Figure 6.1. We assume that the order of creating the threads in each run of the SUT will be deterministic. The better matching of the threads in two separate runs is part of the future work.

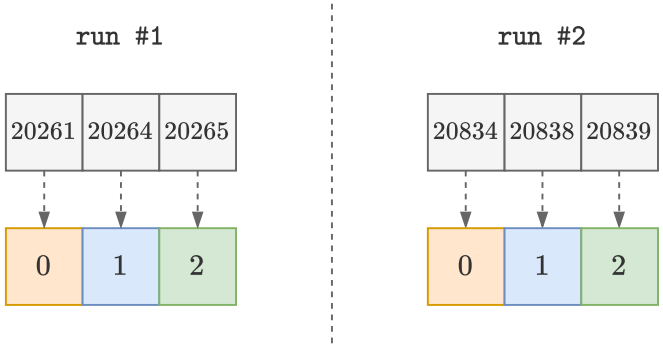

Figure 6.1: An example of mapping the real tids to their substitutes, in order to allow the comparison of the threads in two separate runs.

6.1.3 Extending the Corpus

The corpus can be further extended by other functions, based on the baseline runtime data from the previous section. First, we compute the average runtime $\overline{t_f}$ of each measured function f as follows:

$$\overline{t_f} = \frac{f\_times(f)}{f\_calls(f)} \tag{6.1}$$

Then, we are able to filter the functions according to the *minimal function runtime* value (1000 microseconds by default), which can also be specified by user using the option `--min-func-time`. Note that in the speed-up mode, we form the corpus just with respect to this criteria (and also user-specified corpus). We assume that the longer running functions are better candidates for speed-up simulation and in the case of shortly running functions, only a small amount of noise is then injected into other functions, which can be actually smaller noise than the instrumentation overhead of noise injecting (thus the results are skewed). The user can print the information about functions and the average runtime of the functions (by using `--show-functions-w-time`). An example of such output is depicted in Listing 12.

```
ALL FUNCTIONS (204):
  _ZN3re211FilteredRE2C1Ev <==> re2::FilteredRE2::FilteredRE2() ==> 71.21
  _Znwm <==> operator new(unsigned long) ==> 1878.12
  __tls_get_addr <==> __tls_get_addr ==> 799.67
  _ZN3re26Regexp7DestroyEv <==> re2::Regexp::Destroy() ==> 2049.71
  _ZN3re24ProgC2Ev <==> re2::Prog::Prog() ==> 136.51
  ...
CORPUS (168):
  _Znwm <==> operator new(unsigned long) ==> 1878.12
  _ZN3re26Regexp7DestroyEv <==> re2::Regexp::Destroy() ==> 2049.71
  ...
```

Listing 12: Example output of using `--show-functions-w-time` option.

6.1.4 Building the Pintool for Noise Injection

The Pintool we implemented for the noise injection (the so-called `NoiseInjector` Pintool) is based on the `Tracer Pintool`. Hence, it includes collecting the runtime data and injecting the noise. We repeat that noise has these three parameters in total: the noise location, the noise type, and the noise strength. These parameters could be defined in the `config.h` file, but we want to repeatedly modify the parameters in each perfblowing loop, which would requires building the `NoiseInjector` Pintool in each iteration. Since the Pin allows us to pass the arguments to a Pintool, we decided that the noise parameters will be passed as the `NoiseInjector` Pintool arguments. At the beginning of this Pintool, it parses the given arguments and sets them to the global variables, so they are globally accessible during the analyses. This approach allows us to build the `NoiseInjector` Pintool in advance, and skip compiling before every perfblowing experiment. In conclusion, we show a pseudocode of the perfblowing initialisation phase in Listing 13.

```
# yield functions from SUT binary
f_selector = FunctionSelector(pin_mode_fs)
all_f, corpus = f_selector.run(binary)
# yield baseline data using Tracer Pintool
tracer = TracerPintool(pin_mode, all_f)
baseline_data = tracer.run(binary)
# extend the corpus
corpus_addition = filter_functions(all_f, baseline_data, min_func_time)
corpus.extend(corpus_addition)
# build the Pintool for noise injection
noise_injector = NoiseInjector(pin_mode, all_f)
```

Listing 13: Pseudocode of the perfblowing initialisation phase. The algorithm first builds and executes the `FunctionSelector` Pintool, which analyses binaries and returns all functions that are executed within SUT's `main` function (`all_f`) and information about the functions' calls. These are necessary to decide whether a function met selection criteria conditions and be added to the so-called `corpus`. The next step is to collect the baseline data which are further necessary for evaluating the noise injection. Baseline data are derived from the `Tracer` Pintool result, i.e. time elapsed within every function called from the SUT's binary `main`. Before starting the perfblowing loop, we build the `NoiseInjector` Pintool, which is based on `Tracer` Pintool extended by a noise injection. The build is performed only once, before the loop, and the noise parameters are passed to it at startup as its arguments. The initialisation phase also contains other minor steps (e.g. generating the configuration of the Pintools, creating a *noise database* for slow-down mode, or inserting initialisation values for graphical visualisation), which we did not mention, but we will return to them in the following sections.

6.2 Perfblowing loop

After the initialisation, we can start the loop of the perfblowing experiments. We defined the perfblowing experiment as a tuple (f, t, s, m), such that if: (1) $m = \textit{slow-down}$ the noise is injected at function f of noise type t and with strength s, and (2) if the $m = \textit{speed-up}$ the noise is injected in all other functions except the f. The perfblowing loop consists of choosing the values of the experiment tuple by running the experiment itself and evaluating the results. The pseudocode of the perfblowing loop with its components is depicted in Listing 14.

6.2.1 Selecting a Candidate

In each iteration we select a candidate f, i.e. the function that we want to inject noise into (slow-down mode), or a function that will not be noised, but all the others will (speed-up mode). Each function is scored by its *fitness* value, initialised to 0 at the beginning. During the perfblowing, fitness value of the function is refined according to the results of the perfblowing experiments. We calculate the fitness value equally in both slow-down and speed-up modes, but based on different *indicators*: (1) the *influence* in the slow-down and (2) the *opportunity* in the speed-up mode.

The *influence* of the perfblowing experiment $p = (f, t, s, m)$ denotes a ratio between the so-called $runtime\_degradation(p)$ and $function\_degradation(p)$. The $runtime\_degradation(p)$ is the rate between the *total_runtime* from the results of p and *total_runtime* from the baseline data:

$$runtime\_degradation(p) = \frac{total\_runtime_p}{total\_runtime_b} \tag{6.2}$$

The *function_degradation* value is then defined similarly, as the rate between the time spent in the function f within the p and baseline SUT run.

$$function\_degradation(p) = \frac{f\_times_p(f)}{f\_times_b(f)} \tag{6.3}$$

Therefore, the *influence* value of the p is defined as follows:

$$influence(p) = \frac{runtime\_degradation(p)}{function\_degradation(p)} \tag{6.4}$$

In the speed-up mode, the perfblowing experiment $p = (f, t, s, m)$ has the indicator *opportunity* computed as the sum of the *opt_score* for each thread, where *opt_score* of a thread denotes the ratio between a thread runtime optimisation and function optimisation in p. Although each experiment has a specified noise strength, we cannot say how much of the noise we have actually injected into the SUT (in microseconds)[6]. However, we can approximate the injected amount of noise using the following calculation:

$$noise\_injected\_us(t) = \frac{t\_times_p(t) - t\_times_b(t)}{not\_f\_calls\_in(t)} \tag{6.5}$$

The approximation is based on the difference between baseline thread runtime and noised thread runtime to estimate the total injected noise in microseconds. When we divide the total amount of noise injected in the thread t by the calls of other functions than f within this thread, we can approximate the average amount of noise injected into one function in thread t. For clarity, we will also define the *all_calls(t)* as the number of all function calls within the thread t:

$$all\_calls(t) = f\_calls\_in(t) + not\_f\_calls\_in(t) \tag{6.6}$$

Now we can calculate the estimated speed-up of thread t (in microseconds) as subtraction of two elements: (1) simulation that all called functions were blown-up, i.e. sum of baseline thread runtime and noise in each function call and (2) the runtime of the thread in the experiment p (where f was not blown-up):

$$speedup(t) = (t\_times_b(t) + all\_calls(t) * noise\_injected\_us(t)) - t\_times_p(t) \tag{6.7}$$

Finally, the *opt_score* of a thread t in perfblowing experiment p is computed as the ratio between the *runtime_optimisation(p,t)* and *function_optimisation(p,t)*:

$$runtime\_optimisation(p, t) = \frac{speedup(t)}{t\_times_b(t)} \tag{6.8}$$

$$function\_optimisation(p, t) = \frac{noise\_injected\_us(t)}{\overline{t_f}} \tag{6.9}$$

[6] functions that are used for timed noise guarantees the lower time limit of how long will be the thread suspended, and other noise types implementations do not even have time parameters

$$opt\_score(p,t) = \frac{runtime\_optimisation(p,t)}{function\_optimisation(p,t)} \tag{6.10}$$

Let *tids* be the set of the all thread ids observed during the SUT execution. We can define the *opportunity* value of the experiment p as follows:

$$opportunity(p) = \sum_{t \in tids} opt\_score(t) \tag{6.11}$$

When we characterise the indicators of the perfblowing experiment, which serves as its evaluation value, we can define the *fitness* value of the function f. Let $indicator(p)$ be the indicator value of the perfblowing experiment p (according to the selected perfblowing mode), and P_f be the set of the perfblowing experiments on function f. The *fitness* value of the function f is computed as the average indicator value of the performed perfblowing experiments on the f:

$$fitness(f) = \frac{\sum_{p \in P_f} indicator(p)}{|P_f|} \tag{6.12}$$

After each perfblowing experiment, the fitness value of the selected function is recomputed. When selecting a candidate function for the next perfblowing experiment, we take the fitness values into account. In particular, we divide the sorted corpus functions with respect to their fitness and divide them into five intervals. We derive the number of intervals from [33], because five intervals seem to be appropriate, since with fewer intervals functions are in too big groups and in case of more intervals, functions with similar score are pointlessly scattered. Each interval has assigned a weight w, which is linearly decrementing with the interval index, i.e. the weight of the first interval is the highest: $w = 5$, the weight of the second interval is $w = 4$ and so on. Then we perform a weighted interval selection and randomly select a function from this interval. The intention behind this idea is to select a function with a lower fitness value, so we will do more perfblowing experiments with them if the impact on performance has not been sufficiently enforced yet. The function selection is also depicted in the Figure 6.2.

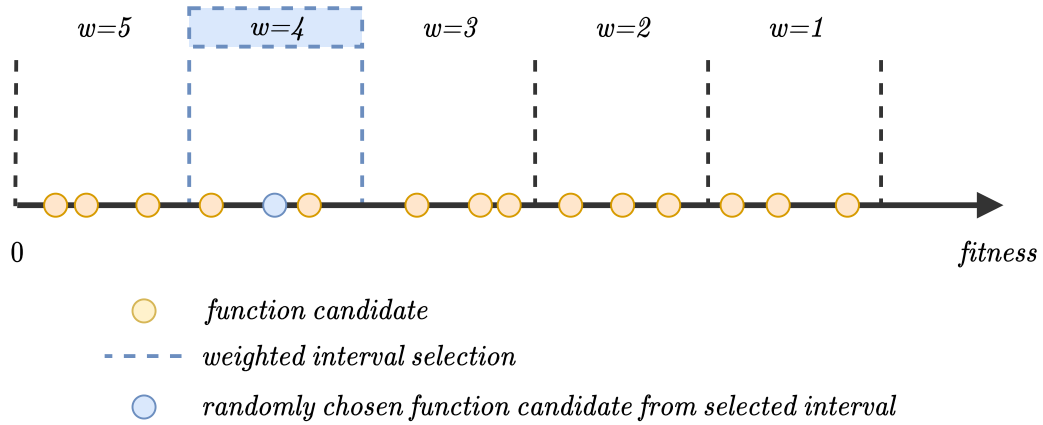

Figure 6.2: An example of selecting a function from the corpus. First we assign weights to the intervals of the sorted corpus, and then select an interval with the respect to the weights (in this example, interval with weight $w = 4$). Finally, we select randomly from the previously selected interval.

6.2.2 Defining the Remaining Noise Parameters

In the previous section, we showed how to select where to inject the noise (i.e. the noise *location* parameter). For this location we now have to determine the noise *type* and noise *strength*. This section describes how we infer those parameters inside the perfblowing loop.

As we discussed in Section 3.3, Pin offers several noise types. The strength of each of these types is represented as follows:

- `nanosleep`: number of nanoseconds of sleep,
- `usleep`: number of microseconds of sleep,
- `PIN_Sleep`: number of milliseconds of sleep,
- `PIN_Yield`: number of subsequent yields of the processor,
- `busy waiting`: number of iterations in busy waiting loop.

The implementation of selecting the noise type and noise strength varies on the selected perfblowing mode. We generate an ordered list of the noise instances[7] within the slow-down mode, which we previously referred to as *noise database*. The list begins with the `nanosleep` noise type and strength defined by *minimal timed noise* converted to nanoseconds. In each iteration we multiply this strength by `growth index` constant. If the strength exceeds the value defined as *maximum timed noise*, the noise type changes to `usleep` and the strength is again set to *minimal timed noise* in microseconds. We continue to iterate over all combinations of noise type and strength. The values *minimal timed noise* and *maximal timed noise* are used for the noise types: `nanosleep`, `microsleep`, `PIN_Sleep`; while for the noise type we instead use `PIN_Yield` it is *minimal yield noise* and *maximum yield noise* constants, and for the noise type `busy waiting` we use *minimal busy noise* and *maximal busy noise*. All these boundaries, as well as the *growth index*, can be set using the framework options. In conclusion, the order of the combination of type and strength of noise is as follows:

(1) (`nanosleep`, *minimal timed noise*[ns]), ..., (`nanosleep`, x_1[ns]),
where $x_1 *$ `growth index` $>$ *maximal timed noise* [ns],

(2) (`usleep`, *minimal timed noise*[us]), ..., (`usleep`, x_2[us]),
where $x_2 *$ `growth index` $>$ *maximal timed noise* [us],

(3) (`PIN_Sleep`, *minimal timed noise*[ms]), ..., (`PIN_Sleep`, x_3[ms]),
where $x_3 *$ `growth index` $>$ *maximal timed noise* [ms],

(4) (`PIN_Yield`, *minimal yield noise*), ..., (`PIN_Yield`, x_4),
where $x_4 *$ `growth index` $>$ *maximal yield noise*, and

(5) (`busy waiting`, *minimal busy noise*), ..., (`busy waiting`, x_5),
where $x_5 *$ `growth index` $>$ *maximal busy noise*.

However, in slow-down perfblowing mode, we want to simulate the speed-up of the function f by x microseconds by injecting the noise of amount x in all other functions. The speed-up, i.e. the injected noise, must be a portion of the $\overline{t_f}$ since f can be actually sped-up at

[7] a noise instance is one combination of noise type and strength, e.g. (`usleep`, 1000)

most by $0 - 100\%$. Thus, we are limited to timed types of noise. We decided to use `nanosleep` because it has the smallest resolution, and we assume it should inject the most accurate amount of desired noise.

After injecting the noise, we approximate the injected noise, w.r.t. the Equation 6.5. The portion of the t_f which will be injected is incremented according to the *sampling* value, which can also be customised using `--sampling` option (10% is default). For example, if the we want to examine function f and we use 10% sampling, first we inject the noise with strength equal to $t_f * 0.1$, in next iteration the noise with strength $t_f * 0.2$ and so on until we get to 100%, i.e. strength equal to t_f.

The framework also offers to use the `-full-analysis` option, which forces that each function in the corpus will be subjected to a perfblowing experiment with each noise type and noise strength.

6.2.3 Perfblowing Experiment

This section covers the core of the perfblowing: **running the perfblowing experiment** and **analysing the perfblowing experiment outcome**. The `NoiseInjector` Pintool is built and all the noise parameters are selected, therefore we are ready to conduct perfblowing experiment. In this part, we execute an external command: `pin -t noise_injector.so -s {strength} -t {type} -l {location} -- SUT_binary [args] [workload]`, where `strength`, `type`, and `location` denote the noise parameters, and `args`, `workload` denote SUT arguments and workload defined by user. Since the `NoiseInjector` Pintool is built upon the `Tracer` Pintool, the output file has the same format. We gather the same information about the perfblowing experiment as in the step **obtaining baseline runtime data**.

Depending on the perfblowing mode, we distinguish two experiment indicators: *influence* and *opportunity*. The influence of the perfblowing experiment p is defined (Equation 6.4) as a ratio between the *runtime_degradation(p)* and *function_degradation(p)*. The intuition behind is that the experiment is scored higher if it triggers greater runtime degradation compared to the function blow-up.

Let us suppose that $p_1 = (f_1, t_1, s_1, m)$ and $p_2 = (f_2, t_2, s_2, m)$ are perfblowing experiments, where p_1 blows f_1 so f_1 takes $2\times$ more time and the SUT runtime degrades by 30%, and $p2$ blows f_2 to f_2 takes $4\times$ more time and the SUT runtime degrades by 20%. Intuitively we can say, that p_1 shows that f_1 has bigger influence, since:

$$influence(p_1) = \frac{runtime\_degradation(p_1)}{function\_degradation(p_1)} = \frac{1.3}{2} = 0.65$$

$$influence(p_2) = \frac{runtime\_degradation(p_2)}{function\_degradation(p_2)} = \frac{1.2}{4} = 0.3$$

The example is also illustrated in Figure 6.3, where the influence value denotes the slope of the line, i.e. how fast does the line grow. The intuition is that the steeper the slope the greater influence the function has on program runtime performance. The *runtime_deg* denotes overall degradation of runtime and *function_deg* denotes degradation of function runtime.

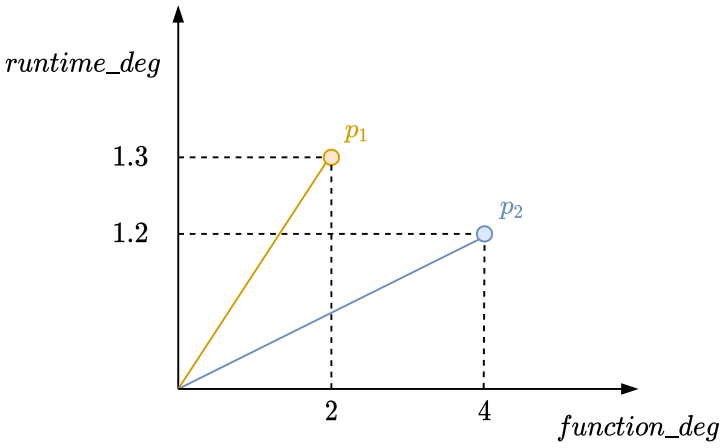

Figure 6.3: An illustration of the example with two experiments and their influence values.

Analogously, using the speed-up perfblowing mode, we compute *opportunity* as an indicator of perfblowing experiment p. The opportunity is defined as a sum of *opt_score* of each SUT's thread, where *opt_score(p,t)* denotes the ratio between *runtime_optimisation(p,t)* and *function_optimisation(p,t)* (Equation 6.10). Again, let us suppose that $p_1 = (f_1, t_1, s_1, m)$ and $p_2 = (f_2, t_2, s_2, m)$ are perfblowing experiments, where p_1 simulate speed-up of f_1 by 20%, p_2 simulate speed-up of f_2 by 40%, and that SUT run consist of only one thread t, for simplicity. We can intuitively say, that the p_1 represents better optimisation opportunity, since:

$$opportunity(p_1) = opt\_score(p_1, t) = \frac{runtime\_optimisation(p_1, t)}{function\_optimisation(p_1, t)} = \frac{0.1}{0.2} = 0.5$$

$$opportunity(p_2) = opt\_score(p_2, t) = \frac{runtime\_optimisation(p_2, t)}{function\_optimisation(p_2, t)} = \frac{0.05}{0.4} = 0.125$$

The particular opt_score of the experiments p_1, p_2 (as the slopes of lines) within the thread t is depicted in the Figure 6.4. The greater line slope determines an experiment with the better opt_score, i.e. p_1 represents better optimisation opportunity.

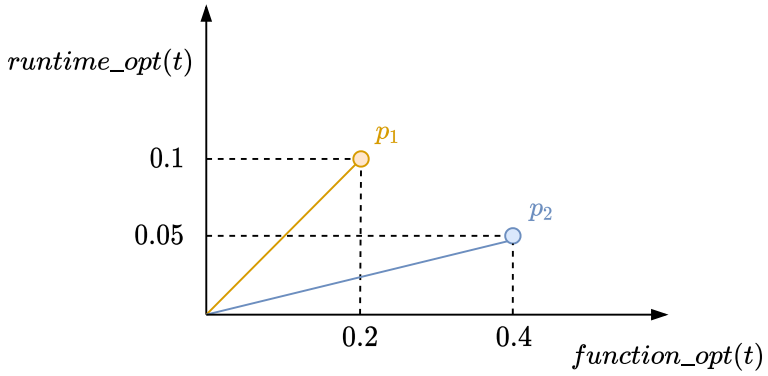

Figure 6.4: An illustration of the example with two experiments and their opportunity values.

Once we compute the indicator values of the perfblowing experiment, the fitness value of the selected function is refined according to Equation 6.12. As last, we store the data of the performed experiment for the purposes of further interpretation. The summary of the perfblowing loop is written in pseudocode in Listing 14.

```
while conditions_to_exit_not_met:
    # select function from the corpus
    function = select_candidate(corpus)
    # get remaining noise parameters
    noise = function.get_noise(perfblowing_mode)
    # run noise injector with SUT binary
    data = NoiseInjector.run(binary, function, noise, perfblowing_mode)
    # handle theresults
    results = parse_data(data, perfblowing_mode)
    indicator = evaluate(results, baseline_results, perfblowing_mode)
    # recompute the fitness value of the function
    function.refine_fitness(indicator, perfblowing_mode)
    # add results to stats
    stats.add(results)
```

Listing 14: Pseudocode of the perfblowing loop. The conditions of leaving the perfblowing loop are either: (1) the specified timeout has been exceeded, or (2) the specified number of the loop iterations has been executed, or (3) all corpus functions were fully analysed, i.e. with each noise type and each noise strength. At the beginning, we select a function which will be subjected to the perfblowing experiment (based on their fitness value), and select the remaining noise parameters. Then we run the `NoiseInjector` Pintool together with the SUT binary and specified noise, which will be injected. Resulting output file is parsed and experiment results are compared to the baseline results, in order to evaluate the perfblowing experiment using indicator value. At last, the indicator value is used for fitness refinement of selected function and the results are added to the stats so at the end we will be able to interpret them.

6.3 Interpretation of Results

This section will describe the last step of the interpretation of the collected data from the perfblowing loop. The result of each perfblowing experiment is recorded to the stats so we can interpret them at the end.

6.3.1 Tabular Interpretation

The first type of interpretation is a textual overview of the achieved results. We provide a table with the top 50 functions sorted by their fitness value (i.e. the average influence or opportunity value of experiments conducted on the function). Moreover, we include the number of experiments conducted on the function, the portion of the time that SUT spent in the function, and the number of function calls (both values are from the baseline SUT run). An example of the tabular interpretation of the perfblowing can be seen in Listing 15.

id	Function	Fitness	Exp	Time %	Calls
1	setlocale	0.87232	2	64.7929	1
2	fclose	0.16963	4	10.4438	2
3	__sysconf	0.10954	3	9.4378	12
4	_IO_file_close_it	0.08945	6	6.0059	2
5	getenv	0.07531	2	6.7455	29
6	__default_morecore	0.05112	4	2.7514	2
7	__strdup	0.03892	3	3.2248	15
8	__libc_free	0.03885	6	3.0473	5
9	__cxa_atexit	0.03516	5	2.1301	2
10	__getpagesize	0.03188	5	2.7218	13
...					
41	__fxstat64	0.00988	8	0.7396	3
42	read	0.00966	6	0.5918	2
43	mmap64	0.00908	15	0.5917	2
44	fileno	0.00838	3	0.4142	2
45	_IO_unsave_markers	0.00673	7	0.4437	2
46	_IO_file_sync	0.00642	12	0.4437	2
47	__fpending	0.00629	15	0.4142	2
48	brk	0.00598	11	0.4437	2
49	_IO_setb	0.00589	17	0.4142	2
50	_IO_default_finish	0.00575	9	0.4142	2

Listing 15: Example tabular interpretation of perfblowing results.

6.3.2 Timeline Graphs

The second interpretation of our framework is the graphical representation of the executions of functions during a SUT run. We choose the perfblowing experiment with the highest indicator value, and plot the data from the run in the form of Gantt chart. The x-axis represents microseconds elapsed from the start of the measuring and the y-axis represents the runs of individual threads. An example timeline graph is in Figure 6.5. All generated graphs are created using the `plotly`[8] package. Moreover, the graphs are interactive (e.g. allows zoom in/out or shows data on hovering on particular data).

6.3.3 Causal Profiles

The other graphical interpretation of perfblowing results is causal profile. In this work, we distinguish two types of causal profiles: (1) influence causal profile: illustration of how the SUT runtime will degrade if we blow-up a function by a certain amount, and (2) optimisation causal profile: demonstration of how much we would optimise the SUT runtime if we optimise a function by a certain amount. The influence causal profile corresponds with the Figure 6.3, only groups the experiments of one function together. Analogously, optimisation causal profile (or profiles, depends on how much thread a SUT run involves) is outlined in Figure 6.4, where we also group all the experiments on a function together. Examples of causal profiles are shown in Figures 6.6 and 6.7.

[8] plotly — https://plotly.com/python/

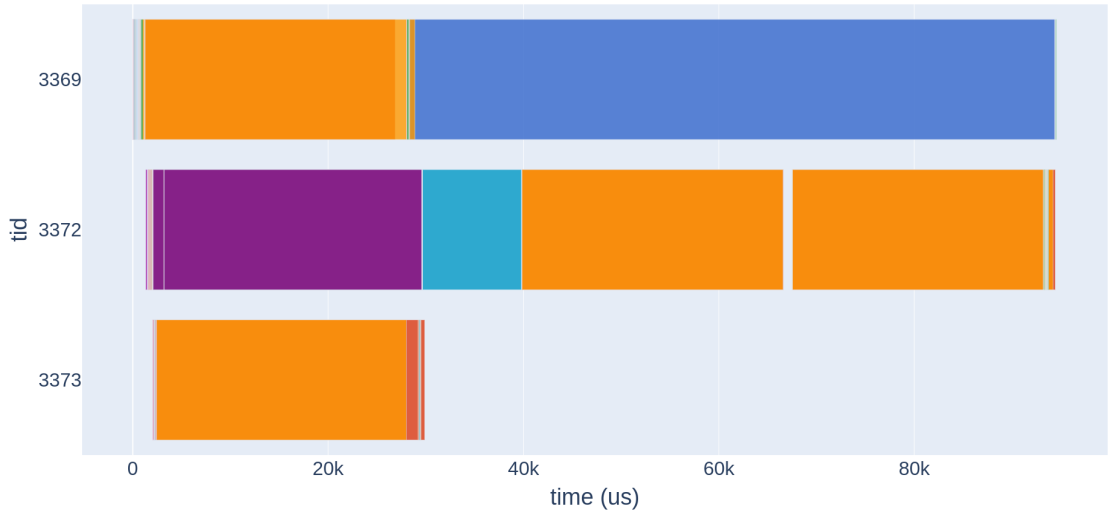

Figure 6.5: An example of a timeline graph from one of the perfblowing results. Each rectangle represents a function call within a certain thread. The orange rectangles denotes the function we blow-up and the functions represented by blue and purple rectangles are waiting for other thread to finish their job. The legend of the graph listing all the functions is omitted from this picture.

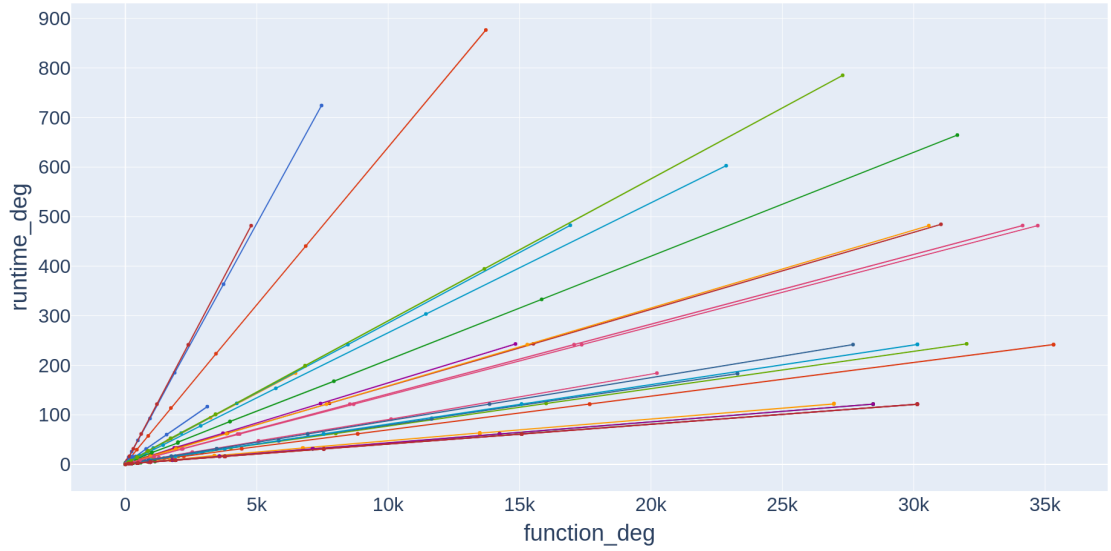

Figure 6.6: An example of an influence causal profile. The graph demonstrates the dependence of degradation of overall runtime on degradation of particular functions for various perfblowing experiments. The more rapidly the dependence grows, the more likely the function is a bottleneck.

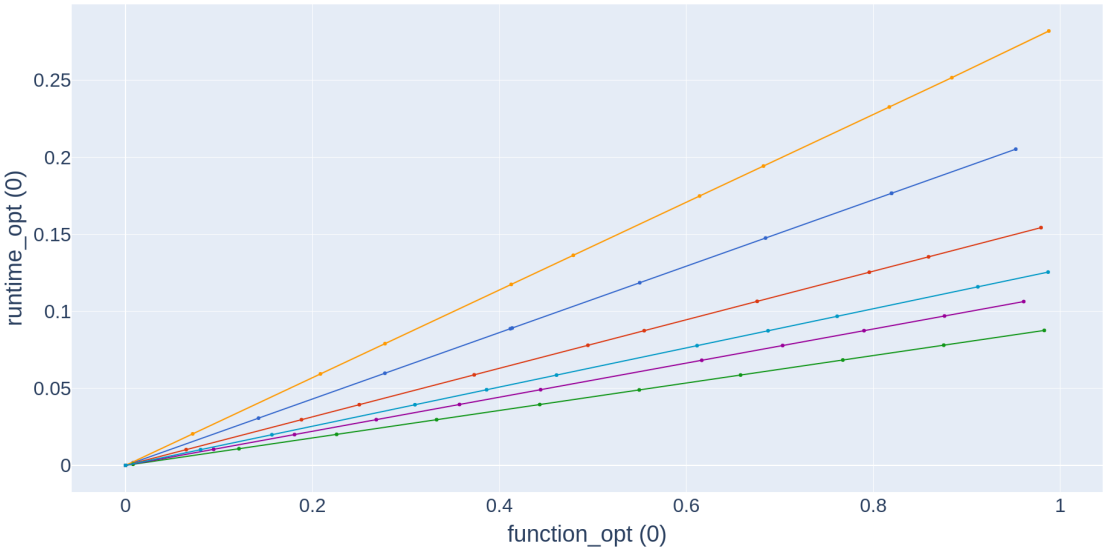

Figure 6.7: An example of an optimisation causal profile. The graph describes how optimisation of a function improves the runtime of a thread (with tid mapping to 0 in this case). Based on this causal profile, the user may observe the effect of the eventual optimisation of the functions.

Chapter 7

Experimental Evaluation

We evaluated the proposed Perun-Blower on two case studies and used it for locating performance bottlenecks as well as for estimating the speed-up of possible optimisations. This chapter analyses the degradation and optimisation impact of functions in two selected projects: `google/re2` library for regular expressions and `Z3` theorem prover.

Machine Specification. We conducted the experiments on a reference machine with the following specification:

Machine	Dell Precision 5510
OS	Ubuntu 19.10 64-bit
Arch	x86_64
Cache	128KiB L1, 1MiB L2, 8MiB L3
CPU	Intel Core i7-6820HQ CPU @ 2.70GHz × 8
RAM	16 GiB @ 2133 MHz
SSD	256 GiB SATA 6Gb/s

7.1 google/re2

Our first case study is the perfblowing of the regular expression library `google/re2`[1]. This library is based on finite-state machine and automata theory, in contrast to almost all other regular expression libraries, leveraging the backtracking approach. Google uses the library in its products like *Gmail*, *Google Documents* or *Google Sheets* [5].

The library Wiki[2] lists two basic operators in the library's API: (1) `RE2::FullMatch` that requires the regular expression to match the entire input text, and (2) `RE2::PartialMatch` that matches a substring of the input text, returning the leftmost-longest match. We used as a subject of the perfblowing the file `testinstall.cc` from the library repository, that is purposed for testing the library installation. The file's code contains calling both of the main operators and other functions from the library (code is available in Listing 23).

[1] google/re2 — https://github.com/google/re2
[2] re2 Wiki — https://github.com/google/re2/wiki/CplusplusAPI

Slow-down Evaluation

The first experiment is the slow-down perfblowing with the following configuration:

Options	
`perfblowing mode`	*slow-down*
`iterations`	*1500*
`Pin mode`	*Probe*
`main function`	*main*
`function selection criteria`	*calls, call-sites*
`minimal function runtime [us]`	*1 000*
`timed_noise_min [us]`	*1 000*
`timed_noise_max [us]`	*1 000 000*
`yield_noise_min`	*1 000*
`yield_noise_max`	*1 000 000*
`busy_waiting_noise_min`	*100 000*
`busy_waiting_noise_max`	*100 000 000*
`noise_growth_index`	*2*
Derived properties	
`all_f`	*204*
`corpus`	*168*

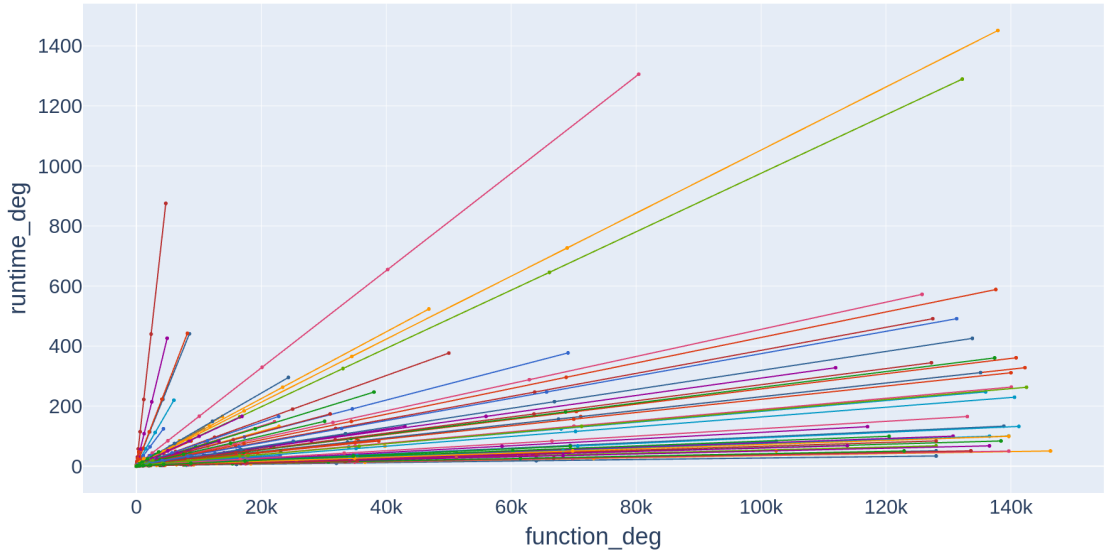

Figure 7.1: Influence causal profile of the experiment on `google/re2`. The graph's legend is not included since there were up to 154 functions in a corpus; each experiment is represented by dot. The long lines which grow at a small angle represent the functions with small influence. The perfblowing algorithm tries to do more experiments with these functions to enforce the manifestation of the possible bottlenecks However, we are more functions represented by steep lines (i.e. the functions with the greater fitness from the results in Listing 21), that denotes that small relative function degradation led to significant runtime degradation.

The tabular output of the perfblowin in Listing 21 lists top 50 functions according to the *fitness* value. Figure 7.1 presents the *influence causal profile* from this testing, which express how a relative function degradation impact the program runtime.

However, we were more interested in the functions that grow most rapidly along with the ratio of their blown-up. From the experiment output (Listing 21), we selected top 19 functions (with *fitness* value higher than 0.3) and performed *full analysis*, i.e. each function was blown up with each noise type and strength. The configuration is the same as in previous measurement, with exception of the corpus (top functions from the previous experiment). The results are in Listing 16 and in Figure 7.2.

id	Function	Fitness	Exp	Time %	Calls
1	re2::RE2::Init(re2::StringPiec...	0.74046	55	54.7954	3
2	re2::Compiler::Compile(re2::Re...	0.64937	55	44.6896	4
3	re2::Regexp::CompileToProg(long)	0.62105	55	38.5375	3
4	re2::RE2::RE2(char const*)	0.59843	55	32.0122	2
5	re2::RE2::DoMatch(re2::StringP...	0.50262	55	25.2343	3
6	re2::RE2::PartialMatchN(re2::S...	0.49060	55	20.4527	2
7	re2::Compiler::Finish(re2::Reg...	0.45397	55	22.9657	4
8	re2::Prog::Flatten()	0.38251	55	16.7598	4
9	re2::Regexp::Parse(re2::String...	0.29232	55	9.3870	3
10	re2::DFA::SearchFFT(re2::DFA::...	0.28342	55	6.7746	2
11	re2::Regexp::Walker<re2::Frag>...	0.28323	55	10.0455	4
12	bool re2::DFA::InlinedSearchLo...	0.28024	55	6.7159	2
13	re2::RE2::~RE2()	0.27904	55	8.7205	3
14	re2::Regexp::Simplify()	0.25068	55	9.4620	5
15	__once_proxy	0.22033	55	8.4076	6
16	pthread_once	0.21551	55	9.2045	8
17	re2::Prog::MarkSuccessors(re2:...	0.17423	55	4.6821	4
18	re2::DFA::~DFA()	0.16128	55	4.4539	4
19	re2::Prog::IsOnePass()	0.14701	55	3.0736	3

Listing 16: Text output from the second experiment on the `google/re2` library where we focused on the top 19 functions from the previous experiment. With the *full analysis* type of the perfblowing, we injected noise of each type and strength to each corpus function. Therefore, we performed 55 perfblowing experiments with each listed function. Value in *Time* and *Calls* columns denote how much of the program runtime the function took and how much was the function called in the baseline testing. As one can notice, the highest influence value has the `re2::RE2::Init` function, in which the program spent over 54% of its time. This initialisation function includes parsing and compiling the regular expression, which is, indeed, a time-consuming task. In this case, fitness value 0.74046 says that if this function took 10× more time, the program runtime would degrade approximately 7.4×. Moreover, in this Listing, we can see other interesting functions, e.g. `re2::DFA::SearchFFT`, which does not take much time resources (~6.77%), but its influence value is relatively high (over 0.28). This function is a specialised version of function `DFA::InlinedSearchLoop` (where the FFT denotes the true/false values used as the `DFA::InlinedSearchLoop` parameters), which represents the generic search loop, that searches text for a match.

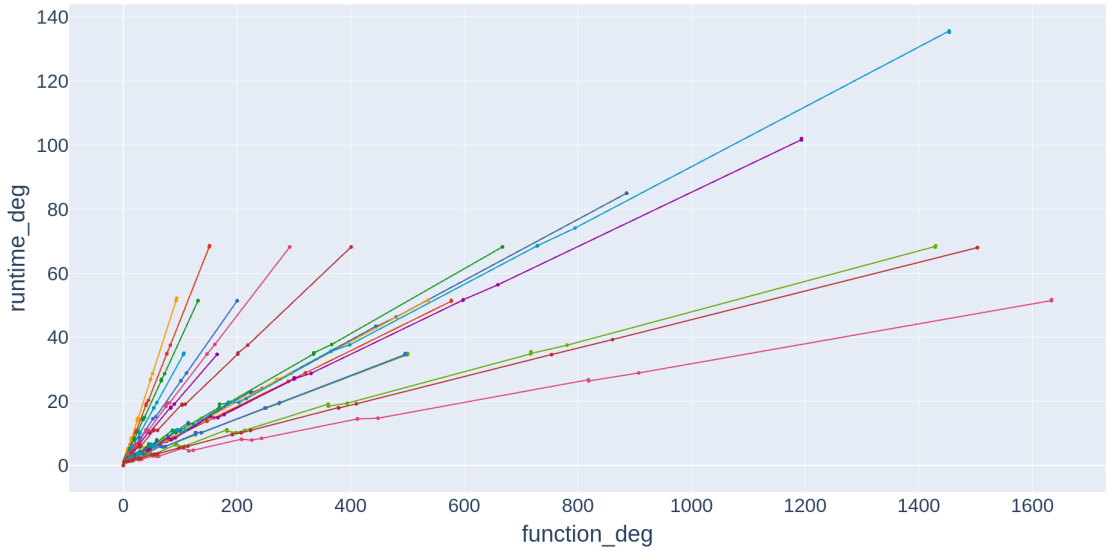

Figure 7.2: Visualisation of the second slow-down experiment with the `google/re2` library. More growing functions have greater influence, as listed in the results of this experiment in Listing 16. Note that we blow up a function maximally by ~1600×, compared to the first experiment with this library, where the blown up exceeded 140 thousand. The reason is that we focus on the smaller amount of functions, and these functions take more portion of the program runtime (in comparison to the functions of the first experiment); thus, their relative degradation (*function_deg*) is smaller, but the impact on the total runtime is significant.

Speed-up Evaluation

Next, we evaluated the speed-up mode of Perun-Blower on this SUT and workload. We empirically set the *minimal function runtime* to 5000 microseconds to focus on the top 10 longest-running functions and thus skip perfblowing of short=running functions. The configuration of this experiment is the following:

Options	
`perfblowing mode`	*speed-up*
`Pin mode`	*Probe*
`main function`	*main*
`minimal function runtime [us]`	*5000*
`sampling [%]`	*10*
Derived properties	
`all_f`	*204*
`corpus`	*10*

The results of this experiment are available in Listing 17. This results show that the implementation should be correct since, in this case, only one thread was started, so the optimisation potential (i.e. fitness value) should directly depend on how much time SUT spent in a given function.

id	Function	Fitness	Exp	Time %	Calls
1	re2::RE2::Init(re2::StringPiec...	0.54941	10	54.9414	3
2	re2::Compiler::Compile(re2::Re...	0.52840	10	52.8404	4
3	re2::Regexp::CompileToProg(lon...	0.40913	10	40.9131	3
4	re2::RE2::RE2(char const*)	0.32281	10	32.2814	2
5	re2::RE2::DoMatch(re2::StringP...	0.28968	10	28.968	3
6	re2::RE2::PartialMatchN(re2::S...	0.24766	10	24.766	2
7	re2::FilteredRE2::Add(re2::Str...	0.22868	10	22.8682	1
8	re2::RE2::RE2(re2::StringPiece...	0.22773	10	22.7738	1
9	re2::FilteredRE2::Compile(std:...	0.06703	10	6.7034	1
10	re2::Prefilter::FromRE2(re2::R...	0.06048	10	6.0483	1

Listing 17: Tabular results of the speed-up perfblowing of `google/re2` functions. The *fitness* value correlates with the time portion that SUT spent in the function, which was exactly what we expected. For example, in the extreme case if we optimised the function `re2::RE2::Init` by 100% (i.e. we would skip the function) we should save the ~54% of the SUT runtime: based on Equation 6.10 we can compute the *runtime_optimisation* as $opt\_score * function\_optimisation$, which gives us $0.54941 * 100 = 54.941$. (notice that we can treat $fitness$ as the *opt_score*, because each experiment with function f should have approximately the same *opt_score* in one-threaded SUT.)

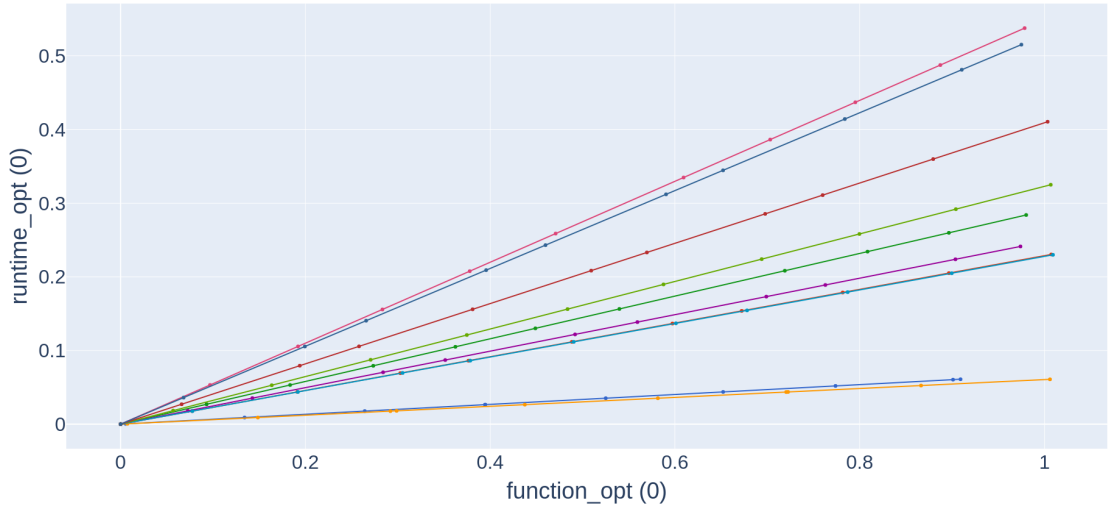

Figure 7.3: Optimisation causal profile from the experiment on `google/re2` library. The more growing lines represent functions with higher optimisation impact on the total runtime of the SUT. The individual lines correspond to the resulting order in Listing 17, thus, for example, the two steepest lines represent experiments on functions `re2::RE2::Init` and `re2::Compiler::Compile`, while experiments with a pair of the last functions `re2::FilteredRE2::Compile` and `re2::Prefilter::FromRE2`, are the least increasing lines. Based on this profile, the user can consider attempts of optimising a function.

7.2 Z3 Theorem Prover

Z3[3] is a state-of-the art theorem prover developed by Microsoft Research. The prover can be used to check the satisfiability of logical formulas over one or more theories. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, data types, uninterpreted functions, and quantifiers. It can be applied, e.g., for static checking, test case generation, or predicate abstraction [7].

To evaluate perfblowing with Z3, we have to provide an input (workload), i.e. the so-called *Z3 script*. Z3 script is a sequence of commands (e.g. command `declare-const` that declares a constant of a given type). We chose a simple Z3 script from the Microsoft Research tutorial[4], shown in Listing 18.

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (> (+ (mod x 4) (* 3 (div y 2))) (- x y)))
(check-sat)
```

Listing 18: Z3 script used for evaluation.

Moreover, we tested the Z3 in parallel solving mode. By setting `parallel.enable=true` as an argument, we force Z3 to spawn a number of worker threads proportional to the number of available CPU cores (8 according to the machine specification).

Slow-down Evaluation

Compared to evaluation of `google/re2`, the run of Z3 with the specified argument for parallel solving and listed Z3 script as its workload, includes much more function calls. Therefore, the output file of each perfblowing experiment is quite large[5] and parsing it is more time consuming.

Hence, we limit ourselves to 500 iterations of the perfblowing loop and just one criterion (*call-sites*) for the function selection to reduce the size of the corpus. Also, we slightly modified the experiment configuration in order to reduce the experiment runtime. The maximal noise values were set to lower values, and we also increased the growth index so the noise strength could rise more quickly. We measure the data of the Z3 functions in the scope of function `__libc_start_main`, since the tested binary does not involve default `main` function. The complete configuration of the first evaluation on Z3 theorem prover is as follows:

[3] Z3 Theorem Prover — https://github.com/Z3Prover/z3

[4] Programming Z3 — https://theory.stanford.edu/~nikolaj/programmingz3.html

[5] output file contains ~32k records with Z3 case study, compared to ~6.3k records with `google/re2` case study

Options	
`perfblowing mode`	*slow-down*
`iterations`	*500*
`Pin mode`	*Probe*
`main function`	*__libc_start_main*
`minimal function runtime [us]`	*1 000*
`function selection criteria`	*call-sites*
`timed_noise_min [us]`	*1 000*
`timed_noise_max [us]`	*50 000*
`yield_noise_min`	*1 000*
`yield_noise_max`	*50 000*
`busy_waiting_noise_min`	*100 000*
`busy_waiting_noise_max`	*5 000 000*
`noise_growth_index`	*3*
Derived properties	
`all_f`	*260*
`corpus`	*97*

The tabular interpretation of the perfblowing results containing a list of top 50 functions is in Listing 22, while the graphical form of results, i.e. influence causal profile, in Figure 7.4.

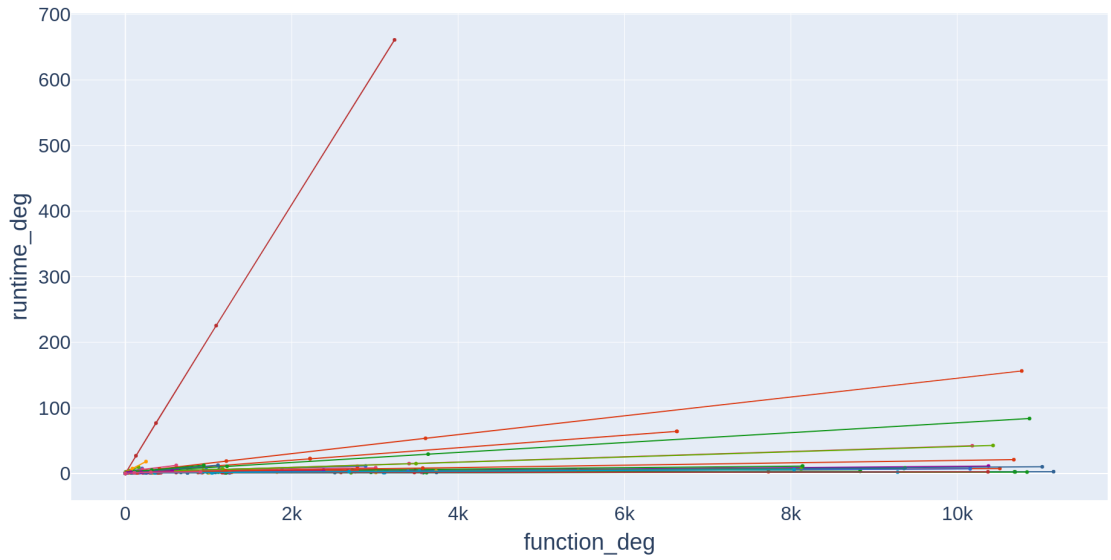

Figure 7.4: Influence causal profile from the first evaluation of `Z3`. Most functions have low influence, and with the growing function blow-up, the runtime does not degrade much. Some functions grew more quickly, but they were selected only in a few perfblowing experiments. These functions are represented by short lines which rise under a greater angle. The most outstanding representative of these functions is `__libc_free` (the long steep line).

Similarly to evaluation on `google/re2`, we performed follow-up *full analysis* experiment, with smaller set of functions. Here we selected top 13 functions (with the influence value higher than 0.2) from the previous experiment.

However, during the perfblowing we observe that program hangs[6] when the certain amount of noise is injected into one of these three functions: `wait*`[7], `__lll_lock_wait`, and `pthread_cond_wait`. We present the noise type and strength which triggered the hangs in Table 7.1.

Table 7.1: Table of `Z3` problematic functions with the detail about the injected noise which led the program to hang. We also provide additional information about how much time portion of total runtime the function took and how many times the function was called. These data are from the baseline run from the first evaluation results, included in Listing 22.

Function	pthread_cond_wait	__lll_lock_wait	wait*
Noise type	nanosleep	nanosleep	nanosleep
Noise strength	27×10^6 (27 ms)	27×10^6 (27 ms)	81×10^6 (81 ms)
Time [%]	62.711	3.6833	63.1441
Calls	7	21	7

Hence, we performed the *full analysis* experiment omitting these functions. Other than that, the experiment has the same configuration as was stated in the first attempt. We present the results of the second perfblowing evaluation on `Z3` in Listing 19 and Figure 7.5.

```
 id  Function                                Fitness    Exp    Time %    Calls
---  ----------------------------------  ---------  ---  --------  -------
  1  std::locale::_Impl::_Impl(unsi...       0.61394   25    3.9184        1
  2  std::ctype<wchar_t>::ctype(uns...       0.49758   25    2.5799        1
  3  std::ctype<wchar_t>::_M_initia...       0.48877   25    2.5620        1
  4  __GI___pthread_timedjoin_ex             0.44234   25    8.6937        8
  5  __pthread_once_slow                     0.43283   25    4.0150        3
  6  pthread_join                            0.43196   25    8.7295        8
  7  std::thread::join()                     0.42156   25    8.7663        8
  8  __libc_free                             0.20948   25   20.3869     8216
  9  std::basic_ios<char, std::char...       0.20843   25    5.4144       41
 10  std::condition_variable::notif...       0.05250   25    0.0734        2
```

Listing 19: Output from the second experiment on the `z3`: top 10 functions, sorted by the *fitness* value. We observe many functions with low time percentage of the program runtime have quite a but big influence. For example function `std::locale::_Impl::_Impl(unsigned long)` is the best scored function with fitness saying that if this function will take 10× more time, the program runtime will be slowed down ~6.139× in average. The function is followed by two functions from `std::ctype` class, with quite big performance impact considering their time-consumption. The rest of the functions have smaller fitness value in spite that they consumed quite much time. However, these functions were also more called, and the fitness value is computed concerning the relative function degradation; thus, we regard the average function runtime.

[6] The program does not exit within several hours (e.g. we keep running the SUT for 3 hours without the program exit).

[7] We replaced function `std::condition_variable::wait(std::unique_lock&)` with `wait*` to save space.

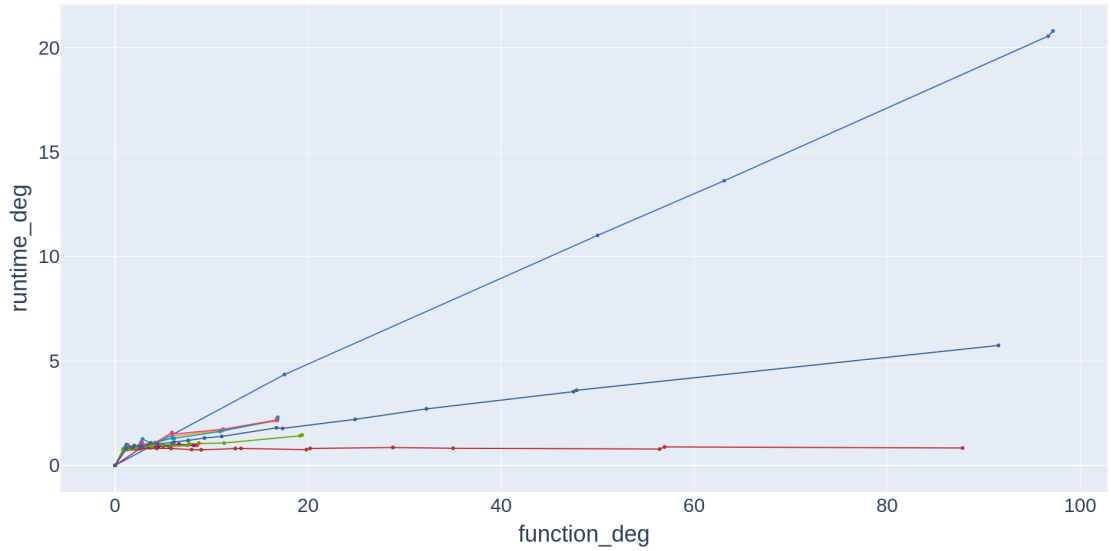

Figure 7.5: Influence causal profile of the second experiment on `Z3`. The three longest lines represent experiments with the functions `std::condition_variable::notify_all`, `std::basic_ios<char, std::char_traits<char> >::init`, and `__libc_free`, from the evaluation Table 19. The lines outstand because, compared to others, one execution of such function took a small amount of time. The blue line represents perfblowing experiments with the `__libc_free` function, which, however, does not have the best fitness. Higher scored functions achieved higher overall runtime degradation while injecting less noise; thus, the influence of these experiments was greater than in the case of the `__libc_free` function. In the graph, we do not present the results of performance experiments with function degradation value over 100 for the sake of presentation.

Speed-up Evaluation

To test the speed-up perfblowing on `Z3`, we had to lower the value of the minimal function runtime to 1000 microseconds in order to obtain a corpus with nine longest-running functions within the SUT. We list the results in Listing 20.

Options	
`perfblowing mode`	*speed-up*
`Pin mode`	*Probe*
`main function`	*__libc_start_main*
`minimal function runtime [us]`	*1000*
`sampling [%]`	*20*
Derived properties	
`all_f`	*260*
`corpus`	*9*

Examining the timeline graph of the best perfblowing experiment observed (which is one of the results interpretation), we found that the program run starts with the main thread (mapping to the tid 0), which later spawns eight more threads (which corresponds with the number of CPU cores of the reference machine).

In addition, from this graph, we extract the data about which of the corpus functions are called in which threads to understand the perfblowing results better. This information is included in Table 7.2.

Table 7.2: Function calls within the program threads. The table cells mark whether the function represented by its id from Listing 20 is called from within the program thread. One can observe that functions with id 1-3 were called in threads 2-8, functions 4-10 were only called within the main thread, and thread 1 did not call any corpus functions.

function id \ *tid*	0	1	2	3	4	5	6	7	8
1	✘	✘	✔	✔	✔	✔	✔	✔	✔
2	✘	✘	✔	✔	✔	✔	✔	✔	✔
3	✘	✘	✔	✔	✔	✔	✔	✔	✔
4	✔	✘	✘	✘	✘	✘	✘	✘	✘
5	✔	✘	✘	✘	✘	✘	✘	✘	✘
6	✔	✘	✘	✘	✘	✘	✘	✘	✘
7	✔	✘	✘	✘	✘	✘	✘	✘	✘
8	✔	✘	✘	✘	✘	✘	✘	✘	✘
9	✔	✘	✘	✘	✘	✘	✘	✘	✘
10	✔	✘	✘	✘	✘	✘	✘	✘	✘

id	Function	Fitness	Exp	Time %	Calls
1	std::condition_variable::wait(...	6.03911	5	50.4574	7
2	pthread_cond_wait	6.01527	5	50.2582	7
3	__pthread_mutex_cond_lock	0.26516	5	2.2154	7
4	std::thread::join()	0.11902	5	11.9027	8
5	pthread_join	0.11851	5	11.8518	8
6	__GI___pthread_timedjoin_ex	0.11804	5	11.8045	8
7	__pthread_once_slow	0.03974	5	3.9740	3
8	std::locale::_Impl::_Impl(unsi...	0.03897	5	3.8977	1
9	std::ctype<wchar_t>::ctype(uns...	0.03004	5	3.0043	1
10	std::ctype<wchar_t>::_M_initia...	0.02987	5	2.9873	1

Listing 20: Results table of the speed-up experiments with `Z3`. The first two functions (used to block the current thread on a condition variable) with the highest fitness value are marked as the best opportunities for optimisation, since their fitness is refined in each perfblowing experiment based on *opt_score* from seven threads in total. This fitness score should tell the user that the optimisation of these functions reduces most of the computational time. In these seven threads, the function `__pthread_mutex_cond_lock` is also called, but it does not take as much time, and so its optimisation will not bring as much benefit as the optimisation of the first two functions. The other functions (4-10) are called only from the main thread, and so their fitness function correlates with the portion of time spent in this them, compared to the total runtime of the main thread.

The second part of the results is a set of optimisation causal profiles, one profile for each thread. However, thread 1 does not involve calling any of the tested functions, and so its causal profile does not give any useful information.

Moreover, the causal profiles of threads 2-8 have almost identical results. Hence, we decided to present a causal profile of the main thread, i.e. thread 0 (see Figure 7.6) and one of the threads 2-8 (specifically thread 2, see Figure 7.7).

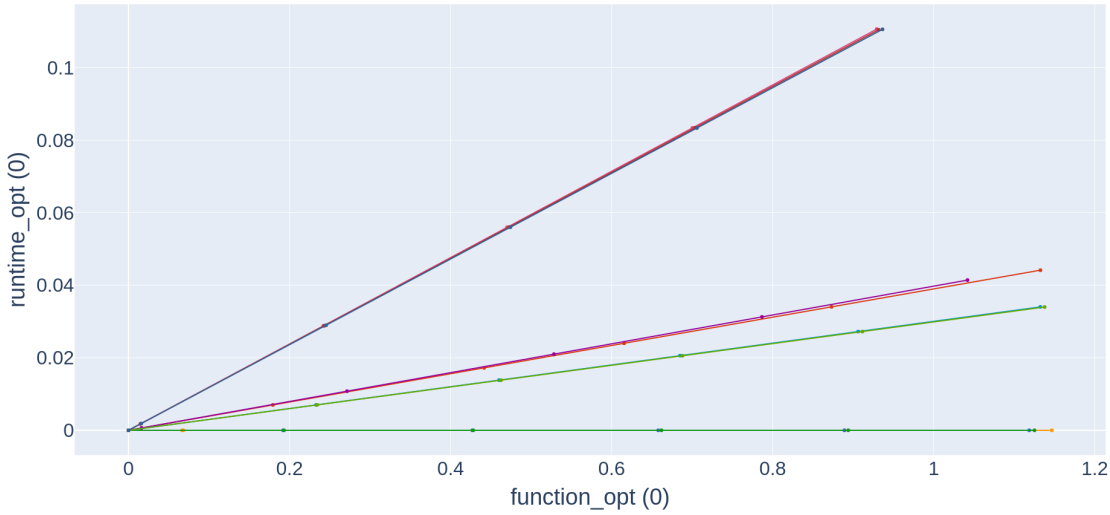

Figure 7.6: Optimisation causal profile of the main thread. The triple of the most growing lines represent the speed-up experiments of the functions `std::thread::join()`, `pthread_join`, and `__GI___pthread_timedjoin_ex`, i.e. the top 3 scored functions called from this thread. We can say from the graph that if we improved any of these functions by ~70 %, we would achieve thread acceleration by~8%. Other lines correspond with the perf-blowing experiments on functions 7-10.

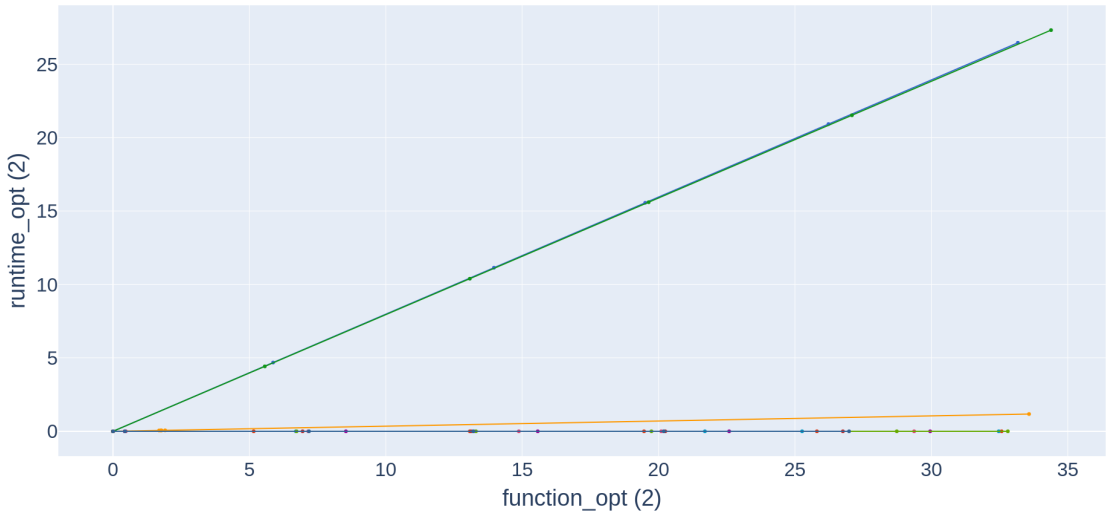

Figure 7.7: Optimisation causal profile of thread 2. Two functions have big optimisation potential, i.e. functions `wait*` and `pthread_cond_wait`. The slope of this line is ~0.85 which means that if we optimise these functions by, for example, 100%, we would save up ~85% of the computational time of the thread. There is also a line representing perfblowing experiments on function `__pthread_mutex_cond_lock`, but with the low increasing, represented by the slope equal to ~0.03.

Chapter 8

Conclusion

In this work, we proposed an algorithm that automatically injects noise into SUT functions and analyses the effect of the noise injection. Noise injection is performed repeatedly: one perfblowing experiment in each iteration of the perfblowing loop. The algorithm is built on the idea of evolutionary algorithms since each of the tested functions is evaluated by its fitness score.

The proposed Perun-Blower framework can work in two modes: (1) slow-down, where noise is injected into the selected function, and (2) speed-up, where we inject noise into all but the selected function. In the slow-down mode, we examine how the degradation of a function (caused by noise injection) affects the performance, while in the speed-up mode, we try to estimate how much the performance of individual SUT threads improves if the selected function were optimised. Basically, we simulate the theoretical optimisation of a function by slowing down all other functions.

The proposed algorithm was integrated within the Perun framework and evaluated on two case studies: `google/re2` regular expression library and `Z3` theorem prover. The evaluation results show that this framework can find SUT functions that significantly impact overall program runtime. During the evaluation on the project `Z3` we even found three functions, which, when blown up, caused SUT to hang, i.e. the SUT did not exit even after a few hours.

Future Work. One of the improvements could be a more sophisticated refinement of the noise type and strength in the perfblowing loop. Another work is to improve the framework to gradually remove functions from the corpus in slow-down mode, which do not affect performance much. Finally, we could implement better mapping of threads from the two runs of SUT in speed-up mode, where we would not have to assume that the thread creation is deterministic.

Bibliography

[1] *BPF Compiler Collection (BCC).* [Online; visited 26.3.2021]. Available at: https://github.com/iovisor/bcc.

[2] *bpf(2) — Linux manual page.* [Online; visited 24.1.2021]. Available at: https://man7.org/linux/man-pages/man2/bpf.2.html.

[3] *eBPF iovisor webpage.* [Online; visited 24.1.2021]. Available at: https://ebpf.io/.

[4] *Linux Extended BPF (eBPF) Tracing Tools.* [Online; visited 24.1.2021]. Available at: http://www.brendangregg.com/ebpf.html.

[5] *RE2: a principled approach to regular expression matching.* [Online; visited 6.5.2021]. Available at: https://opensource.googleblog.com/2010/03/re2-principled-approach-to-regular.html.

[6] *The 5 Most Common Performance Bottlenecks.* [Online; visited 12.3.2021]. Available at: https://www.apica.io/blog/5-common-performance-bottlenecks/.

[7] *Z3 Theorem Prover.* [Online; visited 8.5.2021]. Available at: https://en.wikipedia.org/wiki/Z3_Theorem_Prover.

[8] *Pin 3.17 User Guide.* November 2020. [Online; visited 27.3.2021]. Available at: https://software.intel.com/sites/landingpage/pintool/docs/98314/Pin/html/index.html.

[9] Ahn, M., Kim, D., Nam, T. and Jeong, J. SCOZ: A system-wide causal profiler for multicore systems. *Software: Practice and Experience.* Wiley Online Library. 2020.

[10] Aitel, D. An introduction to spike, the fuzzer creation kit. *Presentation slides, Aug.* 2002, vol. 1.

[11] Benavides, Z., Vora, K. and Gupta, R. DProf: distributed profiler with strong guarantees. *Proceedings of the ACM on Programming Languages.* ACM New York, NY, USA. 2019, vol. 3, OOPSLA, p. 1–24.

[12] Boushehrinejadmoradi, N., Yoga, A. and Nagarakatte, S. A parallelism profiler with what-if analyses for openmp programs. In: IEEE. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* 2018, p. 198–211.

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

[14] Curtsinger, C. and Berger, E. D. Causal Profiling. Citeseer. 2014. Available at: https://popl-obt-2014.cs.brown.edu/papers/profiling.pdf.

[15] Curtsinger, C. and Berger, E. D. Coz: Finding code that counts with causal profiling. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, p. 184–197.

[16] Devor, T. Pin: Intel's Dynamic Binary Instrumentation Engine. *Presentation slides*. 2013. Available at: https://software.intel.com/content/dam/develop/external/us/en/documents/cgo2013-256675.pdf.

[17] Eddington, M. Peach fuzzing platform. *Peach Fuzzer*. 2011, vol. 34.

[18] Edholm, E. and Göransson, D. *Escaping the Fuzz - Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing*. 2016. Master's thesis. Department of Computer Science and Engineering, Chalmers University of Technology. Available at: http://publications.lib.chalmers.se/records/fulltext/238600/238600.pdf.

[19] Eigler, F. C. *Systemtap tutorial*. 2021. [Online; visited 23.3.2021]. Available at: https://www.sourceware.org/systemtap/tutorial.pdf.

[20] Eigler, F. C. and Hat, R. Problem solving with systemtap. In: Citeseer. *Proc. of the Ottawa Linux Symposium*. 2006, p. 261–268.

[21] Fang, L., Dou, L. and Xu, G. Perfblower: Quickly detecting memory-related performance problems via amplification. In: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 2015.

[22] Fiedor, J., Mužikovská, M., Smrčka, A., Vašíček, O. and Vojnar, T. Advances in the ANaConDA framework for dynamic analysis and testing of concurrent C/C++ programs. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, p. 356–359.

[23] Fiedor, T. *Perun: Performance Version System*. [Online; visited 24.1.2021]. Available at: https://github.com/tfiedor/perun.

[24] Ganesh, V., Leek, T. and Rinard, M. Taint-based directed whitebox fuzzing. In: IEEE. *2009 IEEE 31st International Conference on Software Engineering*. 2009, p. 474–484.

[25] Godefroid, P., Kiezun, A. and Levin, M. Y. Grammar-based whitebox fuzzing. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008, p. 206–215.

[26] Google. *Honggfuzz*. [Online; visited 17.2.2021]. Available at: https://honggfuzz.dev/.

[27] Gregg, B. *BPF Performance Tools*. Pearson Education, 2019. Addison-Wesley Professional Computing Series. ISBN 9780136624585.

[28] Householder, A. D. and Foote, J. M. *Probability-based parameter selection for black-box fuzz testing*. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2012.

[29] Jiří, P. and Stupinský Šimon. *Towards the detection of performance degradation.* In: *Excel@FIT'18.*

[30] labs caca. *zzuf - multi-purpose fuzzer.* [Online; visited 17.2.2021]. Available at: http://caca.zoy.org/wiki/zzuf.

[31] Lemieux, C., Padhye, R., Sen, K. and Song, D. PerfFuzz: Automatically Generating Pathological Inputs. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* New York, NY, USA: ACM, 2018, p. 254–265. ISSTA 2018. DOI: 10.1145/3213846.3213874. ISBN 978-1-4503-5699-2. Available at: http://doi.acm.org/10.1145/3213846.3213874.

[32] Liščinský, M. *Analysis, Modeling and Prediction of Program Performance Based On Recent Testing Techniques.* Brno, CZ, 2020. Project Practise. Brno University of Technology, Faculty of Information Technology.

[33] Liščinský, M. *Fuzz Testing of Program Performance.* Brno, CZ, 2019. Bachelor's Thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/19090/.

[34] Lukan, D. *Pin: Dynamic Binary Instrumentation Framework.* September 2013. [Online; visited 27.3.2021]. Available at: https://resources.infosecinstitute.com/topic/pin-dynamic-binary-instrumentation-framework/.

[35] Memon, A. *Advances in Computers.* Elsevier Science, 2016. ISBN 9780128051696.

[36] Noller, Y., Kersten, R. and Păsăreanu, C. S. Badger: complexity analysis with fuzzing and symbolic execution. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 2018, p. 322–332.

[37] O'Brien, D. Teaching operating systems concepts with SystemTap. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.* 2017, p. 335–340.

[38] Pavela, J. *Efficient Techniques for Program Performance Analysis.* Brno, CZ, 2020. Master's Thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/19092/.

[39] Petsios, T., Zhao, J., Keromytis, A. D. and Jana, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2017, p. 2155–2168.

[40] Pham, V.-T., Böhme, M. and Roychoudhury, A. Model-based whitebox fuzzing for program binaries. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* 2016, p. 543–553.

[41] Pourghassemi, B., Amiri Sani, A. and Chandramowlishwaran, A. What-if analysis of page load time in web browsers using causal profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems.* ACM New York, NY, USA. 2019, vol. 3, no. 2, p. 1–23.

[42] Röning, J., Lasko, M., Takanen, A. and Kaksonen, R. Protos-systematic approach to eliminate software vulnerabilities. *Invited presentation at Microsoft Research*. 2002.

[43] Wang, J., Chen, B., Wei, L. and Liu, Y. Skyfire: Data-driven seed generation for fuzzing. In: IEEE. *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, p. 579–594.

[44] Yoga, A. and Nagarakatte, S. A fast causal profiler for task parallel programs. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, p. 15–26.

[45] Yoga, A. and Nagarakatte, S. Parallelism-centric what-if and differential analyses. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, p. 485–501.

[46] Zalewski, M. *American Fuzzy Lop*. [Online; visited 17.2.2021]. Available at: http://lcamtuf.coredump.cx/afl/.

Appendix A

Evaluation Resutls

id	Function	Fitness	Exp	Time %	Calls
1	re2::RE2::DoMatch(re2::StringP...	0.89799	1	25.703	3
2	re2::Compiler::Compile(re2::Re...	0.82535	4	44.4231	4
3	re2::RE2::Init(re2::StringPiec...	0.81112	6	54.2497	3
4	re2::Regexp::CompileToProg(long)	0.80452	4	38.4734	3
5	re2::RE2::PartialMatchN(re2::S...	0.76469	4	20.7086	2
6	re2::RE2::RE2(char const*)	0.72266	6	31.5303	2
7	re2::Compiler::Finish(re2::Reg...	0.70685	3	22.9833	4
8	bool re2::DFA::InlinedSearchLo...	0.58147	3	7.0099	2
9	re2::Prog::Flatten()	0.51893	5	16.6187	4
10	re2::Regexp::Simplify()	0.51793	2	9.577	5
11	re2::DFA::SearchFFT(re2::DFA::...	0.50007	4	7.0734	2
12	re2::RE2::~RE2()	0.40320	5	8.9491	3
13	re2::Regexp::Parse(re2::String...	0.39603	5	8.9300	3
14	re2::Regexp::Walker<re2::Frag>...	0.37468	5	9.9299	4
15	__once_proxy	0.36957	3	8.4452	6
16	pthread_once	0.34251	3	9.2877	8
17	re2::Prog::IsOnePass()	0.33457	2	3.3698	3
18	re2::Prog::MarkSuccessors(re2:...	0.31026	3	4.8211	4
19	re2::DFA::~DFA()	0.30818	3	4.7177	4
20	__pthread_once_slow	0.27612	6	9.0079	6
21	re2::Regexp::ParseState::PushR...	0.25144	1	2.7276	5
22	re2::Regexp::Walker<int>::Walk...	0.24269	1	1.5482	3
23	re2::Regexp::Walker<re2::Regex...	0.23051	3	6.7396	10
24	re2::Prog::ComputeByteMap()	0.22521	5	4.7527	4
25	re2::Regexp::NumCaptures()	0.21784	3	2.3477	3
26	re2::Prog::~Prog()	0.21255	7	5.9019	4
27	re2::Compiler::AddRuneRange(in...	0.20184	4	6.1499	10
28	operator delete(void*)	0.19341	8	18.5262	427
29	re2::Prog::DeleteDFA(re2::DFA*)	0.19233	4	5.1882	8
30	re2::DFA::RunStateOnByte(re2::...	0.18831	8	11.7293	22

id	Function	Fitness	Exp	Time %	Calls
31	re2::Compiler::Add_80_10ffff()	0.17267	6	3.6718	4
32	re2::Prog::GetDFA(re2::Prog::M...	0.13522	6	2.7276	4
33	re2::DFA::WorkqToCachedState(r...	0.13165	12	8.6026	26
34	re2::Compiler::AddRuneRangeUTF...	0.12804	8	10.4608	18
35	re2::Regexp::ParseState::PushD...	0.12690	4	1.855	4
36	re2::Prefilter::Info::Walker::...	0.11426	7	4.2297	8
37	re2::DFA::AnalyzeSearchHelper(...	0.11048	6	2.1490	4
38	re2::DFA::CachedState(int*, in...	0.10099	5	5.2296	24
39	re2::Regexp::Destroy()	0.08909	7	2.8580	7
40	re2::DFA::AnalyzeSearch(re2::D...	0.08737	9	2.2714	4
41	re2::Regexp::ParseState::DoVer...	0.08033	4	1.1110	4
42	re2::Regexp::ParseState::DoFin...	0.07478	9	1.5768	3
43	std::pair<std::__detail::_Node...	0.06131	7	3.0869	18
44	std::call_once<re2::Prog::GetD...	0.05970	9	1.2017	3
45	re2::DFA::DFA(re2::Prog*, re2:...	0.05729	10	1.4782	4
46	operator new(unsigned long)	0.05714	7	5.3726	430
47	__libc_free	0.05485	7	5.1342	427
48	re2::Prefilter::OrStrings(std...	0.05000	10	1.1079	3
49	re2::Regexp::Decref()	0.04743	9	3.6003	53
50	re2::Prefilter::Info::Literal...	0.04289	6	0.5913	3

Listing 21: Output from the first experiment on the `google/re2` library. Top 50 functions, sorted by the *fitness* value.

id	Function	Fitness	Exp	Time %	Calls
1	pthread_cond_wait	1.40775	2	62.711	7
2	std::condition_variable::wait(...	1.39078	3	63.1441	7
3	std::ctype<wchar_t>::_M_initia...	0.92424	2	3.0967	1
4	pthread_join	0.78475	2	13.6565	8
5	std::locale::_Impl::_Impl(unsi...	0.76970	2	4.0254	1
6	std::ctype<wchar_t>::ctype(uns...	0.67172	4	3.1135	1
7	__GI___pthread_timedjoin_ex	0.63037	4	13.6096	8
8	__lll_lock_wait	0.60616	2	3.6833	21
9	std::thread::join()	0.55777	5	13.7039	8
10	__pthread_once_slow	0.52287	3	4.0940	3
11	std::condition_variable::notif...	0.32617	1	0.2383	2
12	std::basic_ios<char, std::char...	0.22476	3	5.8507	41
13	__libc_free	0.20798	4	20.3355	8216
14	__pthread_disable_asynccancel	0.19898	1	0.5229	8
15	std::basic_ostream<char, std::...	0.17278	5	0.4764	2

id	Function	Fitness	Exp	Time %	Calls
16	std::basic_ios<wchar_t, std::c...	0.16661	3	0.5612	4
17	std::locale::locale()	0.114765	4	4.8401	95
18	pthread_once	0.110318	2	4.4105	130
19	std::basic_ostream<char, std:...	0.107283	5	0.5073	4
20	std::basic_ostream<char, std:...	0.099200	5	1.3568	12
21	std::ios_base::Init::Init()	0.087345	3	6.8115	624
22	std::basic_ostream<char, std:...	0.078411	1	0.4437	19
23	std::__basic_file<char>::close()	0.066223	3	0.0895	2
24	__gxx_personality_v0	0.05952	3	1.4449	59
25	std::basic_istream<char, std:...	0.059355	2	0.6829	26
26	operator delete(void*)	0.056713	5	1.9753	50
27	fwrite	0.055185	3	0.5842	18
28	__pthread_mutex_lock	0.054083	3	3.5298	263
29	std::ios_base::~ios_base()	0.052852	2	0.3621	18
30	__gnu_cxx::stdio_sync_filebuf...	0.046559	5	0.6974	18
31	std::ios_base::_M_init()	0.046383	6	1.1482	45
32	std::basic_ostream<char, std:...	0.043437	4	0.6225	19
33	__gnu_cxx::stdio_sync_filebuf...	0.034160	2	0.5477	38
34	__cxa_end_catch	0.031138	5	0.2416	8
35	clock	0.028089	2	0.0279	2
36	std::ctype<char> const& std::...	0.027608	3	0.5863	43
37	__call_tls_dtors	0.026908	6	0.1820	9
38	_pthread_cleanup_pop	0.025447	9	0.3727	18
39	std::basic_ostream<char, std:...	0.025194	5	0.4478	21
40	std::ios_base::ios_base()	0.023970	4	0.3739	26
41	_setjmp	0.023928	9	0.1994	9
42	mmap64	0.022618	3	0.2345	16
43	std::basic_ostream<wchar_t, s...	0.022557	9	0.0562	3
44	_IO_un_link	0.021870	2	0.0159	2
45	__getpagesize	0.021781	8	0.1918	16
46	__cxa_begin_catch	0.021284	3	0.1063	8
47	std::basic_streambuf<char, st...	0.018628	6	0.2165	17
48	_Unwind_DeleteException	0.018588	6	0.1054	8
49	__sigsetjmp	0.018318	2	0.0780	9
50	__cxa_atexit	0.016739	5	1.4349	646

Listing 22: Output from the first experiment on the `z3`. Top 50 functions, sorted by the *fitness* value.

Appendix B

Source Code for Evaluation

```
// Copyright 2008 The RE2 Authors.  All Rights Reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

#include <stdio.h>
#include <re2/filtered_re2.h>
#include <re2/re2.h>

int main() {
  re2::FilteredRE2 f;
  int id;
  f.Add("a.*b.*c", RE2::DefaultOptions, &id);
  std::vector<std::string> v;
  f.Compile(&v);
  std::vector<int> ids;
  f.FirstMatch("abbccc", ids);

  int n;
  if (RE2::FullMatch("axbyc", "a.*b.*c") &&
      RE2::PartialMatch("foo123bar", "(\\d+)", &n) && n == 123) {
    printf("PASS\n");
    return 0;
  }

  printf("FAIL\n");
  return 2;
}
```

Listing 23: Testing file `testinstall.cc`, on which we conducted perfblowing within evaluation on `google/re2` library.

Appendix C

Storage Medium

`/perun/*` — source code of Perun containing Perun-Blower

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

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

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