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 OF PROGRAMS BASED ON PIN FRAMEWORK

ANALÝZA VÝKONU PROGRAMŮ ZALOŽENÁ NA FRAMEWORKU PIN

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PETER MOČÁRY**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ PAVELA**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Močáry Peter**
Programme: Information Technology
Title: **Performance Analysis of Programs Based on PIN Framework**
Category: Software analysis and testing

Assignment:

1. Get acquainted with the Perun project (performance version system) and the field of software performance analysis.
2. Study available instrumentation frameworks and their application for performance analysis, in particular, for collecting performance data. Focus mainly on the PIN framework.
3. Design and implement a Perun module that collects performance metrics of programs using the PIN framework. Focus on collecting additional data (besides runtime of functions), such as loop metrics or parameter values of invoked functions.
4. Design and implement suitable visualization of the resulting collected data (e.g., waterfall graphs).
5. Demonstrate the solution on at least one non-trivial use-case.

Recommended literature:

- Perun project: https://github.com/tfiedor/perun
- PIN Manual: https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/
- Gregg, B. (2020). Systems Performance, (2nd ed.). Pearson. ISBN: 9780136821694.

Requirements for the first semester:

- Items 1 and 2 of the assignment.

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

Supervisor: **Pavela Jiří, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

Abstract

The goal of this thesis is to extend the Performance Version System – Perun by implementing a new Tracer engine leveraging PIN instrumentation framework. This extension implements basic Tracer functionality and, in addition to that, a recording of function arguments' values as well as basic block run–times. The additional data, along with the visualizations introduced in this thesis, provide the necessary context that simplifies the detection of performance degradation. Besides the PIN framework, the new Tracer engine implements an analysis of debug information in DWARF format (using the python pyelftools library) to gather details about function arguments before the data collection process. The resulting engine was tested on multiple implementations of sorting algorithms and successfully detected the most time consuming functions along with the information about the effect of its parameter value on the functions complexity. Testing the PIN engine on a larger–scale project revealed that, in comparsion to other Tracer engine implementations, the engine performs better or comparably, and produces the correct output.

Abstrakt

Cieľom tejto práce je rozšíriť výkonnostný verzovací system – Perun implementáciou nového Tracer engine využívajúceho inštrumentačný nástroj PIN. Toto rozšírenie implementuje základné funkcie Tracer modulu a zároveň zber argumentov funkcií spolu so zberom dĺžky behu základných blokov programu. Tieto nové údaje spolu s vizualizáciami vytvorenými v tejto práci poskytujú potrebný kontext, ktorý zjednodušuje odhalenie zhoršenia výkonu. Okrem nástroja PIN využíva Tracer engine python knižnicu pyelftools na analýzu ladiacich informácií vo formáte DWARF pre zistenie podrobností o argumentoch funkcií pred procesom zberu údajov. Výsledný engine bol testovaný na viacerých implementáciách triediacich algoritmov a úspešne detekoval časovo najnáročnejšie funkcie spolu s informáciami o zvýšenej zložitosti súvisiacej s jej argumentom. Testovanie na projekte väčšieho rozsahu odhalilo, že v porovnaní s ostatnými implementáciami Tracer engine, tento nový engine pracuje lepšie alebo porovnateľne a produkuje správne výstupy.

Keywords

performance testing, performance bottlenecks, Perun, dynamic instrumentation, PIN

Kľúčové slová

výkonnostné testovanie, výkonnostné úzke miesta, Perun, dynamická inštrumentácia, PIN

Reference

MOČÁRY, Peter. *Performance Analysis of Programs Based on PIN Framework*. Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Pavela

Rozšírený abstrakt

Hlavným cieľom tejto práce je rozšírenie stávajúcej implementácie a funkcionality výkonnostného verzovacieho systému — Perun. Konkrétne sa zaoberá jedným zo zberačov výkonnostných dát, ktorý sa nazýva Tracer. Tracer má za úlohu meranie dĺžky behu jednotlivých funkcií zvoleného C/C++ programu pričom umožňuje realizáciu viacerých oddelených implementácií svojej funkcionality, takzvaných enginov. Implementácia nového enginu pomocou inštrumentačného nástroja PIN je hlavnou časťou tejto práce, avšak, okrem základnej funkcionality, toto rozšírenie nástroja Perun pridáva možnosť detailnejšieho zberu dát. Podporuje zber obmedzenej množiny argumentov funkcií a taktiež dĺžku behu jednotlivých základných blokov kódu zvoleného programu. Tracer okrem zberu dát vykonáva spracovanie zozbieraných dát do jednotného formátu — výkonnostného profilu, ktorý je ďalej používaný na dodatočnú analýzu alebo vizualne spracovanie nazbieraných dát vrámci nástroja Perun. Táto práca sa takisto venuje implementácii vizualizácie nových typov nazbieraných dát, ktoré interpretujú výkonnostný profil užívateľovi za cieľom jednoduchšej manuálnej analýzi vzniknutého výkonnostného profilu.

Na implementáciu nového Tracer enginu bol použitý inštrumentačný nástroj PIN, ktorý podporuje dynamickú binárnu analýzu a zároveň nie je do značnej miery závislý od jadra operačného systému Linux čo umožňuje jeho spustenie bez administrátorských privilégií. Predošlé implementácie Tracer enginov využívajú technológie eBPF a SystemTap, ktoré tieto práva vyžadujú. Využitie nástroja PIN vyžaduje vytvorenie tzv. pintool, ktorý definuje priebeh inštrumentácie. Pintool je možné vytvoriť v jazyku C alebo C++ pomocou API poskytnutej nástrojom PIN. V rámci Tracer enginu sa využívajú rôzne pintooly, čo viedlo k integrácii dynamickej generácie pintoolu pomocou Jinja2 šablón, vďaka ktorým si užívateľ dokáže zvoliť vhodnú konfiguráciu inštrumentácie a v konečnom dôsledku výstupné dáta zozbierané týmto enginom. Zber hodnôt argumentov jednotlivých funkcií vyžaduje, aby engine pri generácii pintoolu poznal názvy funkcií a pozície spolu s typmi parametrov týchto funkcií, ktorých argumenty je nutné zozbierať. V prípade, že si užívateľ zvolí zber argumentov funkcií, engine vykoná analýzu ladiacich infromácií prítomných v poskytnutom binárnom súbore. Ladiace informácie vo formáte DWARF engine analyzuje pomocou python knižnice pyelftools a generuje tabuľku funkcií, ktorých argumenty majú podporovaný typ vhodný pre analýzu. Tieto informácie sú ďalej využité vo vizualizácii vzťahu hodnoty argumentu a dĺžky doby behu funkcie v jednej z implementovaných vizualizácií. Nové vizualizácie vytvorené vrámci tejto práce využívajú python knižnice ako Pandas, Bokeh alebo spojenie Seaborn s Matplotlib.

Experimentálne ohodnotenie vytvoreného Tracer enginu bolo vykonané na viacerých triediacich algoritmoch za účelom preukázania správnosti zozbieraných výsledkov ale aj ich prínosu pri analýze výkonu programov. Experiment zahŕňal správnu a zároveň nesprávnu implementaciu algoritmu, čo Tracer engine správne rozlíšil a vďaka dodatočným informáciám o argumentoch funkcií dokázal odhaliť značné odchýlenie od predpokladanej zložitosti algoritmu. Naviac označil správnu funkciu ako najviac časovo náročnú pričom poukázal na časovo najnáročnejšie základné bloky danej funkcie. Experimenálne bol nový Tracer engine porovnaný s predošlími implementáciami využívajúcimi eBPF a SystemTap. Tento experiment bol vykonaný na projekte väčieho rozsahu, kompresovacom programe CCSDS a ukázal, že engine založený na technológii PIN je rýchlejší alebo porovnateľný s výkonom ostatných enginov a taktiež označil časovo najnáročnejšie funkcie rovnako ako ostané enginy.

Performance Analysis of Programs Based on PIN Framework

Declaration

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

.....
Peter Močáry
May 10, 2022

Acknowledgements

I would like to express my special thanks to my supervisors from VeriFIT performance team Ing. Jiří Pavela and Ing. Tomáš Fiedor Ph.D. for their guidance and frequent consultations providing recommendations and valuable feedback during the course of this Thesis. Furthermore, I would like to thank Ing. Jan Fiedor Ph.D. for providing valuable information regarding the utilized framework.

Contents

Chapter 1

Introduction

Software testing is an essential part of a development process that provides vital information about reliability and quality of the final product. Since testing plays such a crucial role in today's software development, there are tools and techniques incorporating it into the development cycle such as *continuous integration* (CI). CI makes the testing easier and reduces the time it takes to validate and subsequently release new software updates. Even though the software testing techniques are commonly used among developers, the main emphasis is on the software functionality and its performance is often overlooked until the users start noticing problems – which might often be too late.

Performance testing techniques are commonly used by the developers to detect and fix performance issues in software. However, performance testing often poses a bigger challenge than functionality testing, since some performance issues can only be exposed under very specific conditions. Moreover, such conditions may be extremely difficult, or even borderline impossible, to meet. Despite the difficulties, performance testing can improve the quality and user experience of software, e.g., by eliminating all sources of potential slowdowns. The introduction of automation to the performance testing is crucial, because complex programs can produce enormous amounts of data that, when managed without automation, might result in errors. Although the importance of performance testing is undeniable, the variety and quality of available tools is not sufficient. Thus, the lack of monitoring and integration tools focused on performance and its evolution during the development could be a reason for such low interest in performance testing.

The VeriFIT research group developed an open source light weight Performance Version System – Perun [9, 10], which strives to provide the necessary tooling to make the performance testing easier and therefore more utilized by the developers around the world. Perun archives it by integrating Version Control Systems (VCS) and performance regression testing. By creating and storing profiles for each version of a given program, Perun ensures that a developer has better feedback regarding the project performance with every change they make. Performance profiles of the given program are gathered by the *Collectors*, among which the Tracer collector has a major role in measuring the run-times of functions. In its current state, users are able to chose between two different backends (called *engines*) of the Tracer collector based on eBPF [12] and SystemTap [17] frameworks.

This thesis focuses on extending the Perun Tracer collector with a new engine based on the PIN framework while reducing the time spent on the collection of necessary data for

the performance analysis. The PIN framework, compared to the eBPF and SystemTap frameworks currently used in Perun's Tracer collector, offers better options namely in the dynamic binary instrumentation approach, which considerably increases Tracer's potential. Moreover, this extension will allow Tracer to gather additional information regarding function parameters and other code primitives (such as basic block metrics). Such information will be used to evaluate code performance in a more granular fashion, thus leading to more accurate analysis results. Furthermore, this thesis also focuses on visualization of the collected data, as proper visualization greatly reduces the time needed to evaluate the analysis results by the Perun users.

Structure of the thesis. Chapter 2 introduces the Performance Version System – Perun and its architecture while focusing mainly on the Tracer collector. Chapter 3 describes the PIN framework, creation of pintools and compares PIN to the frameworks currently used in the Perun Tracer, highlighting the reasons for this Tracer extension. Chapter 4 covers the requirements of this work and Chapter 5 presents the design and implementation of the Tracer extension and the visualization of collected data. The experimental evaluation of the PIN engine, including the new visual representations of the collected data, is located in chapter 6.

Chapter 2

Perun

This chapter introduces Performance Version System – Perun and argues why Perun is a suitable tool for performance testing. One of the key parts of this chapter is a description of Perun's architecture which sheds light on the internals of Perun, and provides the necessary knowledge to properly understand its workflow. The next section covering Tracer collector is essential for this thesis since it describes the *common engines interface*, which needs to be utilized to extend Tracer with a new *engine* based on the PIN framework, and the data collection process including its strategies as well. The most recent Perun documentation [10] along with the relevant work [27, 26] is utilized in this chapter to provide necessary knowledge for the goals of this thesis.

2.1 Overview

The open-source lightweight Performance Version System – Perun (Performance Under control) was created by the VeriFIT research group to achieve full automation of performance management. Although still under active development, it already has a lot to offer.

Perun works as a wrapper around a VCS and adds support for automated performance testing on top of it. Management of the performance profiles for each version of a project, postprocessing of a created profile and its effective interpretation are other significant features of Perun's tool suite. The automation of a project's performance analysis allows for easy regression testing and the fact that every minor version of the project has its performance profiles stored as a part of development history makes it possible to detect problems associated with performance very early on during the project development, without the need of manual involvement of a user.

Perun is meant to be used by a single developer (or a small team) as a complete solution for storing, automating and interpreting the performance of a project, as well as by bigger teams working on more complex projects. Figure 2.1 illustrates the intended use-case of Perun where each developer keeps his own instance of both versioning and performance systems, and can share the code changes, as well as the performance data, with other developers.

Perun offers a number of advantages over manual management of performance, data such as storing the profiles in a database or directly in a VCS. Since it stores created perfor-

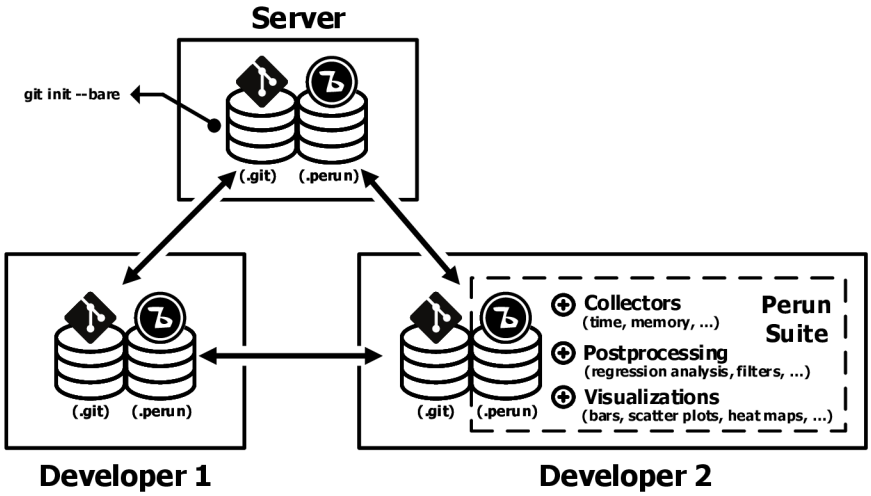


Figure 2.1: An illustration of a project development with Perun deployed in parallel to a VCS (in this case Git) [9].

mance profiles parallel to the VCS and assigns them to a specific version of a project, Perun provides context to the collected data and project's performance history. This allows users to not only identify the origin of a performance issue, but also the optimization of the collection process based on the source code differences from previous versions. Moreover, users will not forget to run a profiling whenever there is a new version of a project, thanks to Perun's automation using the so-called *hooks* in the supported version control systems. Hooks trigger sequences of Perun commands when a VCS action is detected, e.g., whenever there is a new commit. These Perun command sequences are called *jobs* and their specification is inspired by Continuous Integration systems. As Figure 2.2 illustrates, Perun provides a report regarding the discovered performance changes, where each of these changes contains information about its location, severity and confidence. The severity and confidence are supposed to inform a user about the reliability of the detected change alert.

Another Perun's advantage is the genericity of its tools. Currently, the tool suite of Perun contains generic (as well as some specific) visualization, postprocessing and collection modules that form the basic building blocks necessary for specification of jobs and interpretation of collected data. Furthermore, the suite can be extended rather easily with only a few requirements that new modules must comply to. Other than that, Perun provides an easy-to-use interface inspired by the git version control system, offering new users that are, however, experienced with git, not so steep learning curve. Currently, Perun interacts with users mainly through the Command Line Interface (CLI) that fully supports all of its features. A prototype of a Graphical User Interface (GUI) is currently in development.

2.2 Architecture

Perun consists of components that manage the performance profile creation, postprocessing, interpretation, analysis and, as a consequence, provide the users with the necessary tools

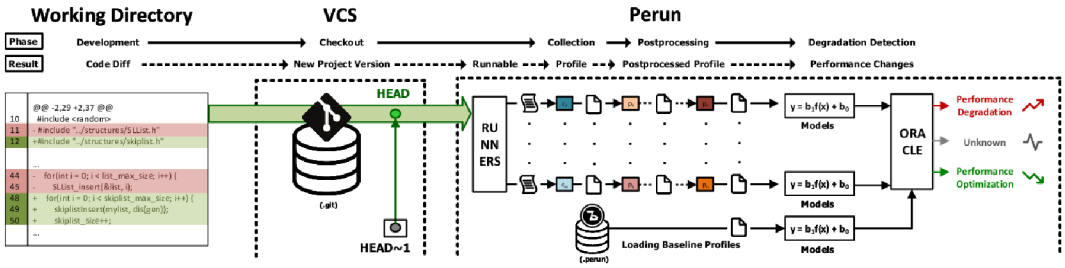


Figure 2.2: This figure describes a workflow of Perun [9]. For each new project version, a number of tasks (known as *jobs*) is collecting performance data, processing it and searching for performance degradation or optimization compared to previous project version.

for robust performance analysis. Figure 2.3 shows that Perun's architecture can be divided into four main logically separate units – data, logic, view and check.

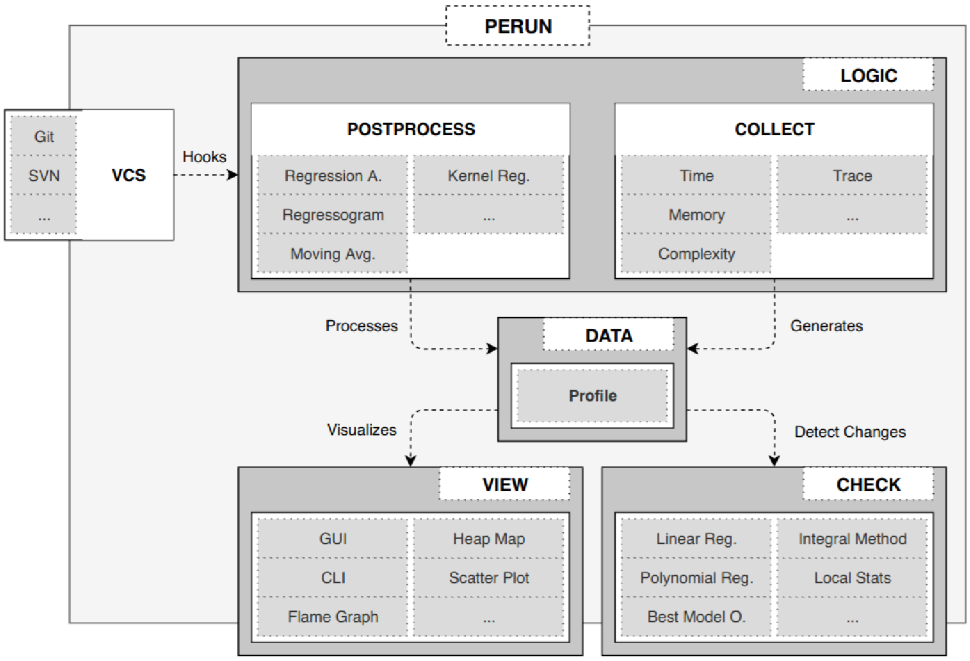


Figure 2.3: The architecture of Perun, as divided into separate units (*data*, *logic*, *view* and *check*) and the VCS module (containing interface for Git, SVN, etc.). The data unit is interacting with every other unit and managing profiles, which are created and processed by the logic unit. The view unit is responsible for interpretation of the collected data and the check unit searches for performance degradation based on the profiles. Taken from [27].

Data. This unit provides an interface for the performance profiles management, which is utilized by every other unit. Because of that, the Data unit represents the core of the architecture. Among the key interface operations is the finalization of a profile form and handling of queries regarding the collected data. Profiles are unified under a format based on JSON which allows a great flexibility for the communication between the Units, and easy extensibility.

Logic. The profiles are created and processed in the logic unit, where the profiling data is gathered and parsed by the *Collectors*, and possibly processed by the *Postprocessors* for further interpretation. This unit also handles many tasks related to the automation, CLI and repository configuration including the VCS hooks. Users can select from multiple collectors based on the information they seek. Perun contains the following collectors:

- *Trace collector* measures the time consumption of functions and custom code blocks. Design of the Tracer architecture allows the user to choose from a range of so-called engines, which utilize different instrumentation frameworks for the collection of performance data. More in-depth description of the Tracer collector can be found in section 2.3.
- *Memory collector* is focused on gathering information about memory allocations in C or C++ programs. The recorded data contain overall heap memory usage with many related attributes, such as memory allocation types or their target addresses. The data collection is facilitated using the `libunwind`[1] library and custom `libmalloc` libraries.
- *Time collector* is implemented as a simple wrapper around the `time` utility and collects the overall duration of arbitrary commands.
- *Bounds collector* performs automated static analysis of worst-case resource bounds of C programs. This collector leverages the `Loopus`[2] tool for computing bounds of loops or Facebook Infer plugin Cost for asymptotic complexity analysis of functions. While `Loopus` is limited to integer programs only, it computes symbolic bounds for each function and loop, highlighting the main source of the complexity. The Bounds collector then reports the complexity of analyzed functions using the big-O notation.

Postprocessors are used for transformation of the data, which helps with identification of potential relations among them. The notable postprocessors currently implemented in Perun are:

- *Normalizer postprocessor* is used for scaling of the collected data to the interval (0,1). This postprocessor is meant to enable profile comparison when the profiles were not created with the same workload or parameters.
- *Regression analysis* offers various computational methods and models for finding fitting models for trends in the captured profiling resources. The regression analysis requires dataset with independent and dependent variables to find a fitting model for dependent variable based on the independent one. The postprocessor currently aims to find a well suited model (linear, quadratic, logarithmic, etc.) for the amount of elapsed time depending on the size of the data structure the function operates on.
- *Regressogram method*, or binning approach, is a simple non-parametric estimator. This method tries to fit models through data by dividing the interval into N parts, where each part is represented by a value equal to the result of the selected statistical aggregation function within the values in the concrete part. The regressogram is

[1] See https://www.nongnu.org/libunwind/.
[2] See https://forsyte.at/software/loopus/.

hence a step function (i.e. constant function by parts). The thesis [29] describes this method, its implementation and also other statistical methods in more detail.

View. This partly independent unit is responsible for input/output interaction with a user. This unit provides a number of visualization techniques that provide better interpretation of the collected data. Some of the currently supported visualization methods are:

- *Bars Plot* is capable of visualizing multiple types of resources as bars while providing the user with a moderate customization possibilities thanks to the Bokeh[3] library, which is used to generate interactive HTML files.
- *Flow Plot* also utilizes the Bokeh library for visualization of the collected data as a flow. This method supports a high number of profile types.
- *Heap Map* can be used to visualize the data collected by the Memory collector. The visualization contains memory address map with representation of memory usage, such as the allocated objects or frequency of the address usage.

Check. Consists of detection methods that report possible changes in performance of a project. Check expects a pair of performance profiles where one represents the new version of the project, and the other represents the *stable* version. These profiles are then compared, which provides relevant information about the state of the new version of the project. Based on the particular resource types present in the profiles, this unit uses various methods, among other the *Average Amount Threshold* method or the *Integral* and *Local Statistics* methods which were introduced in [29].

2.3 Tracer Collector

One of the collectors present in Perun is the Tracer collector. This collector is an important part of the Perun tool suite since it gathers information about run-times of selected functions and custom code blocks executed during the profiling of a program, while keeping track of the call hierarchy as well. Tracer architecture allows multiple implementations of its backend (so-called *engines*). Engines leverage different instrumentation frameworks for the purpose of collecting performance data.

The usage of Tracer can be described in the following manner. A user can select Tracer as the collector through the Perun interface, along with the specification of the collection parameters. Among the parameters are code locations that are going to be measured. These locations are selected either manually by the user, or automatically by Tracer — for which the user can select one of the strategies provided by the Tracer. The collection process itself is divided into four stages: *before*, *collect*, *after* and *teardown*.

1. In the *Before* stage Tracer initializes the selected engine and uses it to instrument[4] the code with handlers for every function or custom code block according to the specified collection strategy.

[3] See https://bokeh.org/ for more information.
[4] Instrumentation is a technique for inserting extra code into an application to observe its behavior.

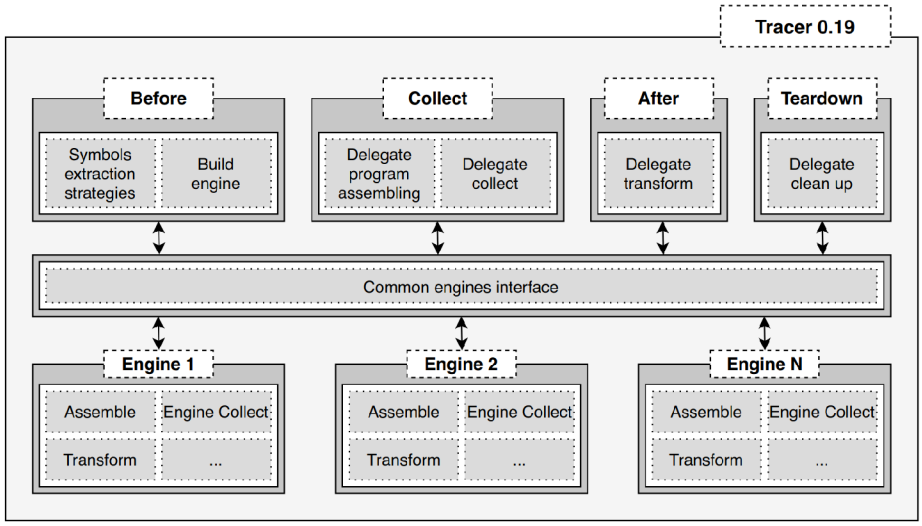


Figure 2.4: Schematic overview of Tracer architecture introduced in version 0.19 in the [27], which unifies interface for multiple engines.

2. The *Collect* stage facilitates the collection of performance data by launching the executable file of the program and tracing it until the process terminates, or a timeout is reached.

3. The *After* stage encapsulates the transformation of the collected raw data output into Perun resource records, which are then stored in a profile.

4. At the end of the collection process, the cleanup of all the used resources (such as temporary files or instrumentation framework processes that are still running) takes place in the *Teardown* stage.

Tracer, as well as any other collector, needs to meet certain requirements to be reliable. Low overhead is a major requirement, because it extends the time period needed for the data collection. Minimization of influence on the collected data, namely the run-time of the *system under test* (SUT) is necessary. Collectors should also minimize the number of dependencies and not require an manual modifications of source code by the user. However, fully satisfying all of the requirements is not possible, therefore finding a balance between speed, accuracy and memory requirements of the collector is crucial.

Each of the Tracer engines must implement the *Common Engines Interface* [27] (see Figure 2.4) that abstracts the communication with concrete engines. This interface allows for easier extension of the Tracer with another engine that leverages a new framework for performance data collection. This approach enables implementation of multiple engines where each of them introduces new advantages over the other and allows a user to decide which engine suits his needs the best. Common Engines Interface is designed as the following set of functions (where $\rightarrow$ represents return type):

- `check_dependencies`: checks that all of the engine requirements are satisfied and all of the dependencies are available.

- `available_usdt` → `dict`: extracts available User-space Statically Defined Tracepoint (USDT) probes, which were defined by the developers of SUT in a framework-specific manner.
- `assemble_collect_program`: assembles the collection program with respect to the specification of profiled probes.
- `engine_collect`: runs the collection process.
- `transform` → `generator`: transforms the raw performance data collected by the selected engine to the unified Perun resources.
- `cleanup`: frees the set of resources that have to be cleaned up in order to avoid serious issues, such as corruption of collected performance data.

The previously mentioned *collection strategies* enhance the automation of Tracer. Every strategy defines what functions should be profiled, without the need to specify them manually. One of the major strategies is the *Userspace strategy* that filters out function symbols that have not been defined by the user, such as various helper functions created by the compilers (`_init`, `_fini`,...). The collection strategies also include the *All strategy*, which instruments all of the functions within the executable file with no filtering whatsoever, or the *Custom strategy* that allows a user to specify the function symbols without utilizing any automatic extraction. The [27] contains information about the collection strategies while also specifying their advantages and disadvantages.

Chapter 3

PIN Framework

Dynamic analysis of programs usually (but not exclusively) requires robust software instrumentation tools for tasks such as profiling, performance evaluation, and bug detection. One of such instrumentation tools is the PIN framework, which can be utilized for program profiling. This Chapter introduces PIN and pintools (Section 3.2), goes over its inner workings and addresses PIN's efficiency and transparency. Section 3.4 briefly introduces eBPF and SystemTap frameworks and, compares them to PIN. Furthermore, the differences between the two modes of instrumentation supported by PIN are discussed in Section 3.3. Other than the introduction of PIN, this Chapter tries to establish the arguments for choosing PIN for the implementation of Tracer's engine. This Chapter uses the information from previous works related to Perun [27, 19].

3.1 Overview

PIN [20] is an instrumentation system for program analysis, developed by Intel, which supports the Linux, MacOS and Windows operating systems and IA-32, x86-64 and MIC instruction set architectures. The goal of PIN is to provide an instrumentation platform for building a wide variety of dynamic program analysis tools with the emphasis on ease-of-use, portability, transparency, efficiency, and robustness. PIN performs Dynamic binary instrumentation of applications at run-time on the compiled binary files. Thus, it requires no recompilation of the source code and can instrument programs that dynamically generate code.

Instrumentation of an application using PIN is done with *pintools* written in C/C++, using PIN's rich application programming interface (API), which allows a pintool to insert calls to handlers at arbitrary locations in the executable file. The API allows access to architecture-specific information and abstracts away the underlying instruction set idiosyncrasies, making it possible to write portable instrumentation tools. The Section 3.2 covers the pintools further.

PIN provides efficient instrumentation by using a just in time (JIT) compiler to insert and optimize code. To further optimize the jitted code, PIN implements code caching, register reallocation, inlining, instruction scheduling and other techniques. This fully automated approach distinguishes PIN from most other instrumentation tools, such as Valgrind [25]

or DynamoRIO [5], which require the user's assistance to boost performance. PIN supports process attaching similar to a debugger — it first attaches to a process, instruments it, collects profiles and eventually detaches. This approach significantly improves the performance since the overhead caused by PIN is present only when attached to a process. The support for attaching and detaching to a process is necessary for the instrumentation of large, long-running applications.

PIN guarantees the instrumentation transparency by preserving the original application behavior which means that the application observes the same addresses and same values as it would in an uninstrumented execution. An example of this behavior could be an application unintentionally accessing data beyond the top of the stack, so PIN will not modify the application stack. The instrumentation transparency makes the collected information more relevant, and is also necessary for correctness of the measured data.

PIN, in its essence, is a just in time compiler, which however, expects a regular native executable instead of a bytecode as its input. Since PIN works on a layer above the operating system (see Figure 3.1), it can only capture user-space code. When an instrumented program is running, there are three binary programs present: the application, PIN and the pintool. PIN is the engine that jits and instruments the application while the pintool contains the analysis and instrumentation routines. Pintool is also linked with a library that allows communication with PIN and they share the same address space, however, they do not share any libraries.

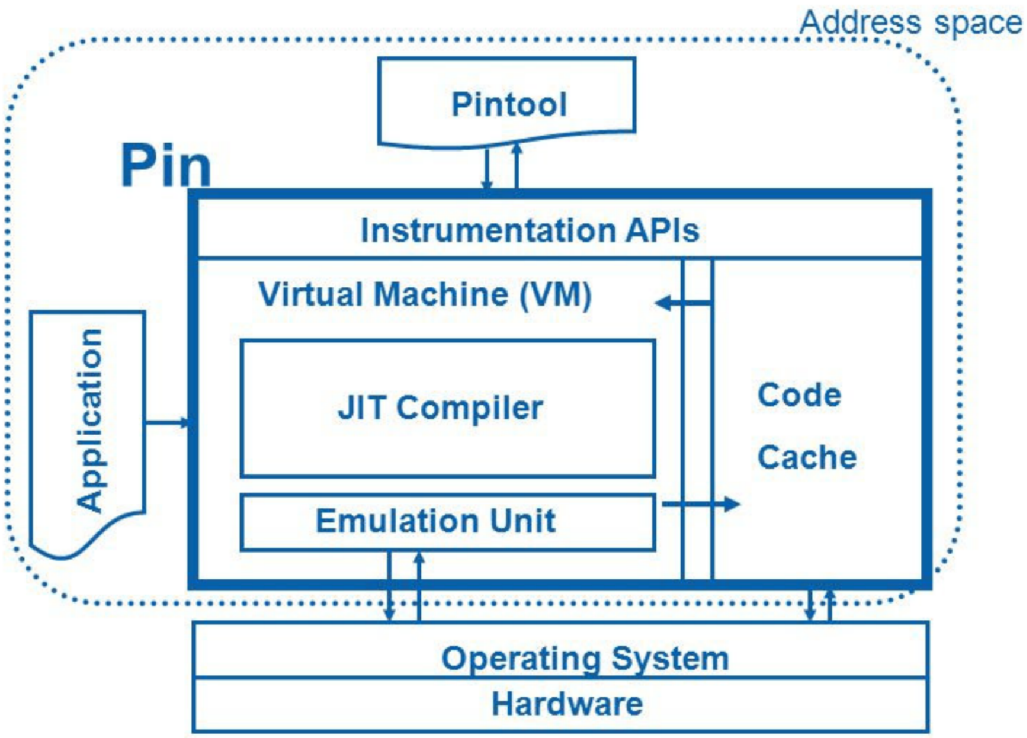


Figure 3.1: The software architecture of the PIN instrumentation framework showing that PIN works on the layer above operating system while reading the selected application and instrumenting its code according to pintool specification. The image was taken from [7].

The execution of a program while using PIN to instrument its code works as follows. PIN intercepts the execution of the first instruction of the executable, generates and compiles new code for the next trace — a straight-line sequence of instructions which terminates at an unconditional control transfer (branch, call or return statements) or when a predefined number of conditional control transfers or individual instructions have been fetched in the trace [20] — starting at this instruction and then passes control to the generated sequence. This newly generated code sequence is almost identical to the original, but PIN ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Every time JIT compiler fetches some code, pintool has the opportunity to instrument the code before its translation. This means that only executed instructions can be instrumented. The instrumented code is kept in memory for its reuse, which also makes PIN more efficient.

3.2 Pintools

The tools specifying the instrumentation details for PIN, called pintools, enable the tool writer to analyze user-space applications. A pintool is a compiled binary file. For Linux, it is a shared library with a `.so` extension and for Windows systems, it is a dynamic library with a `.dll` extension, and dynamic library with a `.dylib` extension for macOS. Pintools can be thought of as plugins that can modify the code generation process inside PIN. PIN allows tool writers to analyze an application at the instruction level without the need of detailed knowledge of the underlying instruction set thanks to the API, which makes the tool writing easier. The API is designed to be architecture independent whenever possible and allows context information, such as register contents, to be passed to the injected code as parameters. This rich API provided by PIN enables instrumentation at these different abstraction levels [21]:

- *Image level* allows the pintool to process an entire image. Thus iterating through the whole program sections, routines in a section or individual instructions in a routine is possible instrumenting the program.
- *Routine level* allows the pintool to process a routine at a time with the possibility of iteration over instructions inside the routine.
- *Trace level* allows the pintool to process one trace at a time by starting from the current instruction and ending with an unconditional branch (e.g. call or return statements).
- *Instruction level* allows the pintool to process an instruction at a time.

Since pintool shares the same address space as PIN and the instrumented executable, that pintool has access to all of the executable's data, even the file descriptors and other process information. Pintools in general have two major components that need to be defined in a pintool:

- *Instrumentation routines* define the precise location where instrumentation is to be inserted (e.g. before or after an instruction).

- *Analysis routines* define what needs to be done when the instrumentation is activated (e.g. increment a counter).

3.3 JIT and Probe Modes

Until now, this chapter presented the JIT mode. In the JIT mode the code that is actually executed is the code generated on-the-fly by PIN and the original code is used only as a reference but never executed. This mode uses JIT compiler to generate instrumented code according to the specification created by user in the pintool. However, PIN can also operate in the so-called Probe mode.

The Probe mode is a method of inserting *probes* at the start of specified routines. The application and the *replacement routine* are run natively in this mode, which improves performance at the cost of putting more responsibility on the pintool writer. A probe is a jump instruction (also called *trampoline*) that is placed at the start of specified routine and redirects the flow of control to the replacement function, which can also call the replaced routine. The Probe mode enables only instrumentation on a routine level, i.e. probes can be placed only at the routine boundaries.

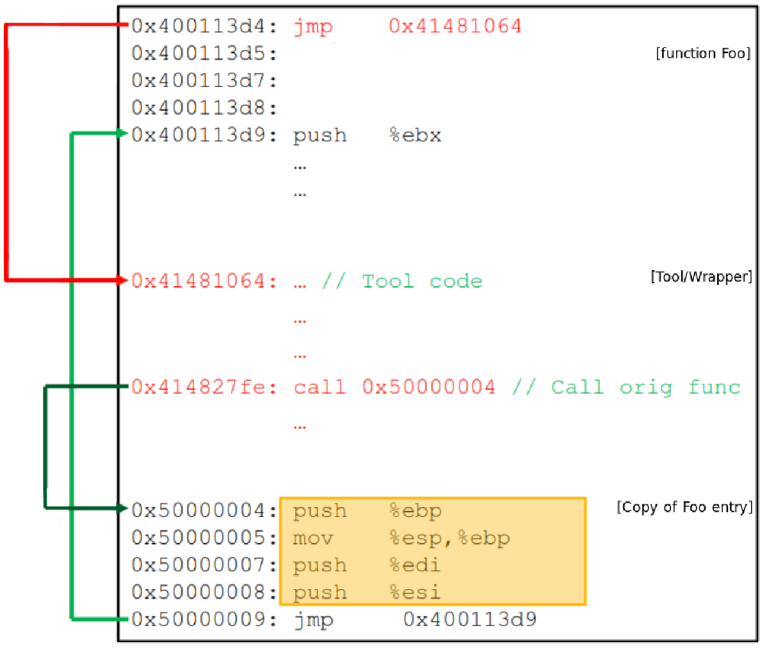


Figure 3.2: A sample probe code for function `Foo`, taken from [8]. This example illustrates instrumentation in PIN Probe mode where the probe itself is placed just before the `Foo` function (at address 0x400113d4). The function is copied to addresses 0x50000004–0x50000008. The probe unconditionally jumps to the instrumented version of the function (defined in a pintool) which also calls the original function.

When comparing these two modes, the JIT mode is far more flexible and common approach and even though it might be slower than Probe mode, it introduces transparency and ease-of-use. The Probe mode has, on the other hand, lower overhead approach that is more

efficient, but also less flexible. The Probe mode also does not provide transparency since the original instructions in memory are overwritten by trampolines.

3.4 Using PIN in Perun's Tracer Engine

Perun (see Chapter 2) has multiple implementations of Tracer engines (discussed in Section 2.3) utilizing different frameworks. This thesis focuses on implementation of a new engine using the PIN instrumentation framework to extend the Tracer with faster dynamic binary instrumentation thanks to the low overhead of PIN framework. To better understand why is this new engine implementation utilizing PIN a beneficial addition to the Tracer, this section briefly introduces the other instrumentation frameworks currently used in Tracer engines and compares them to PIN instrumentation framework.

SystemTap [11, 17] provides an infrastructure for gathering information about the running Linux system (as well as the process running on the system) and enables its detailed analysis. This allows developers and administrators to identify causes of performance issues or bugs. SystemTap is a tracing and profiling framework, which supports all of the current state-of-the-art dynamic instrumentation and probing mechanisms: *kernel tracepoints*, *USDT*, *kprobes*, *uprobes*, *performance counters* and, to a certain degree, in-kernel programming. This makes SystemTap one of the most powerful general-purpose profiling frameworks available for probing of the kernel-space events, as well as the user-space events. The instrumentation is performed by leveraging custom kernel modules to inject probes and their handlers (a code that is executed when a probe is activated). These kernel modules are automatically created from the script provided by the user, which specifies events and defines handlers for them. The script is then translated to the C language to create a kernel module that is subsequently loaded by the system.

eBPF [12, 6] stands for *extended Berkely Packet Filter* — mechanism that makes the Linux kernel dynamically programmable. As Brendan Gregg said in his book *BPF Performance Tools* [12]: "*eBPF does to Linux what JavaScript does to HTML*". Originally designed to capture and filter network packets, eBPF is an highly enhanced version of the original BPF used to filter network packets. The eBPF features extended instruction set and optimizations for modern hardware which opened the possibility to write more complex programs. In its essence, eBPF (also reffered to as simply BPF) is a highly advanced virtual machine (VM) inside the kernel that runs instructions from its own instruction set in an isolated environment. As stated in [6], "*In a sense, you can think of BPF like how you think about the Java Virtual Machine*" — it is a specialized program capable of running machine code, compiled from high-level programming language, inside Kernel. The eBPF engine consists of an interpreter and a JIT compiler that translate the executed eBPF instructions into a native system instructions. The user defined program for eBPF can be supplied at a runtime, and is verified by the *eBPF verifier* which ensures that the provided program will not compromise the user's system by crashing the kernel. The communication between kernel and user-space is done through *eBPF maps* [23]. This design greatly improves the means of dynamic in-kernel programming and allows the user to run custom mini programs in the kernel.

Although both of these frameworks do their job well, there are differences and therefore, each of the frameworks has its own set of advantages and limitations. The SystemTap

framework, for example, provides a multitude of options when probing kernel-space events for a wide range of kernels. However, some kernel versions do not support user-space probing. SystemTap also needs kernel *debuginfo* even though the generic kernel comes with `CONFIG_DEBUG_INFO` and `CONFIG_KPROBES` disabled, which means that some distributions require recompilation of the kernel before running SystemTap. Since both of the frameworks closely cooperate with kernel, they require elevated privileges. This is not required by PIN, because this framework operates in the user-space. Similarly to SystemTap, the eBPF framework requires a fairly recent version of kernel to unlock its full potential. The eBPF framework was developed as part of the Linux kernel and hence is strongly dependent on it, which makes eBPF programs less portable. When it comes to security, eBPF excels, however, the security is forced by limitations on the programs created by the user. While PIN does not rely on the Linux kernel as much as the other frameworks, it relies much more on the processor architecture. PIN also supports multiple operating systems, which favors its portability. Overall summary of the PIN framework compared to other Tracer engine implementations can be found in Table 3.1.

Table 3.1: A comparison of eBPF, SystemTap (both frameworks already used in Tracer engines) and the PIN framework proposed for a new engine implementation. This table highlights the advantages of PIN over the current implementations, and even though PIN does not support the kernel-space instrumentation, the new engine leveraging PIN framework will be a valuable addition to Perun's Tracer.

	SystemTap	eBPG	PIN
Kernel-space instrumentation	✓	✓	✗
Users-space instrumentation	✓	✓	✓
Defines own handlers	✓	✓	✓
Fully dynamic tracing	✗	✓	✓
Does not require root privileges	✗	✗	✓
Does not require kernel debuginfo	✗	✗	✓
Does not rely on recent kernel version	✗	✗	✓

Chapter 4

Analysis of Requirements

This chapter provides an overview of the planned functionality of the resulting Perun extension. The list of functional and non-functional requirements, along with a brief description of the resulting functionality, ensures that the reader is acquainted with not only the capabilities and constraints of Perun's new Tracer engine, but also with the aim of this thesis.

4.1 The Resulting Functionality

This work aims to improve the Perun performance version system by introducing a new useful and reliable extension for one of the existing collectors – Tracer. The new extension should leverage PIN framework to not only implement existing Tracer capabilities (e.g. function run-times collection) for the programs written in C/C++, but also extend the scope of collected data by gathering basic function parameters and run-times of every executed basic block.

The resulting Tracer engine should support a collection of data in two modes: JIT and Probe mode. The JIT mode will be the default mode of the engine and should allow the collection of function run-times as well as certain function parameters and basic block run-times. The Probe mode is also to be fully supported, however, its restrictions do not allow collection of basic block run-times.

The decision of which mode will be used and which additional data to collect will be made by the user of the engine before its execution. Based on the selected settings, the engine will be able to assemble a pintool — definition of how the data is going to be collected and stored. However, the arguments collection process requires an analysis of the debug information in DWARF format [24, 28] to determine the types of arguments and their indices. This work focuses on collecting a basic set of argument types, namely integers, characters and strings (`char*`) while also supporting real numbers (`float` and `double`), although not fully. This set of supported argument types will be extended in future work.

Since the Perun's task in general is to detect performance issues and suggest possible opportunities for optimization, this work will introduce two new visualizations that strive to help with these goals. One of the visualizations should focus on capturing the dependence of function run-times on its arguments values, which helps developers analyze the source of function complexity. The other visualization should display the most time consuming

functions and the basic blocks in these functions with the amount of time consumed by them.

4.2 Functional Requirements

The intended functionality of this thesis and the core requirements for the Tracer engine utilizing PIN instrumentation are as follows:

1. **FR_PE (Perun extension):** The resulting project is integrated into Perun and extends its functionality as one of its Collectors.
2. **FR_TE (Tracer engine implementation):** Implemented in Tracer collector as one of its engines, providing full functionality of a collector.
3. **FR_PIN (PIN framework):** Leverages PIN framework for the collection of the required performance-related data.
4. **FR_PI (Pintool implementation):** Implements multiple pintools that describe how a program should be instrumented.
5. **FR_PM (Pintool Makefile):** Creates general Makefile for compilation of implemented pintools.
6. **FR_JM (Support for JIT mode):** Fully supports PIN's JIT mode for data collection from running applications.
7. **FR_PM (Support for Probe mode):** Supports Probe mode for data collection from running applications with respect to its limitations.
8. **FR_FRT (Function run times):** Is able to collect information about the duration of function execution and other necessary data for its identification.
9. **FR_FAI (Function argument information):** Is able to obtain values from function arguments and extract valuable information form them.
10. **FR_SRF (Support for recursive functions):** Fully supports collection of required data in recursive functions.
11. **FR_BRT (Basic block run-times):** Is able to collect lengths of basic block execution times and other data for its identification.
12. **FR_SMA (Support for multi-threaded/process applications):** Provides support for any multi-threaded/process application and is able to distinguish collected data in each thread/process.
13. **FR_PG (Profile generation):** Converts the collected data to a generic format (i.e. Perun profile) and extends it to encompass the function arguments and basic block run-times.
14. **FR_VIZ (Visualization of collected data):** Provides a user with the ability to visualize the collected data for further manual analysis.

4.3 Non-functional Requirements

While the functionality is a key part of this thesis, the following non-functional requirements play a big role in the overall quality of the final Tracer engine:

1. **NFR_SCA (Scalability):** The new Tracer engine design takes into account that programs using Perun could be of any scale and supports them with reasonable speed.
2. **NFR_MAIN (Maintainability):** Well documented and readable implementation with possibility of easy modifications by independent developers. This is one of the key factors for a long lifespan of a project.
3. **NFR_REL (Reliability):** The reliability of a program must be one of the main priorities of every project. Final implementation is well tested and passes the tests utilized by the Perun Pull Request toolchain.
4. **NFR_MO (Minimal overhead):** The collection of data regarding time is sensitive to any introduced overhead and therefore the implementation focuses on reduction of overhead, not only at collection part of Perun's process.
5. **NFR_MD (Minimum number of dependencies):** The implementation introduces a minimal number of mandatory dependencies. Although a useage of already created and reliable libraries is to be expected, the new Tracer engine will use only necessary and maintained dependencies.
6. **NFR_QUX (Quality of user experience):** The whole process of data collection is automated and provides an easy interface for selecting the desired settings with options for manual analysis of collected data.

Chapter 5

Design and Implementation

This chapter contains a comprehensive description of implementation with emphasis on design decisions. Firstly, this chapter presents details of extending the Perun with the implementation of a new Tracer engine using PIN framework (Section 5.1). Furthermore, this chapter covers implementation details of additional Tracer extensions (Section 5.2) and lastly, Section 5.3 describes the implementation of visualizations of collected data.

5.1 Tracer PIN Engine

To create a basic Tracer engine, which collects data about function run-times, one needs to implement the generic engine interface, which consists of six independent methods. Each of these methods represents a different stage of the data collection process, a phase where different actions need to take place. The introduction to Tracer's engine interface in Section 2.3 briefly describes each method, and the following Section 5.1.1 describes the implementation details regarding the interface.

A substantial part of the engine's collection process is the creation of the collection program, a pintool in this case. The collection program specifies how the collection of data should be executed, and this is the core of the whole collection process. A closer specification of pintools can be found in Section 3.2, and implementation details regarding pintool creation and design, along with the challenges related to its compilation, are described in Section 5.1.2.

The data collected by PIN need to be processed right after the collection into the Perun profile, which unifies the format of data and enables its analysis afterward. On top of that, this part of the process also facilitates the conversion of collected time data from time stamps into actual run-times of every execution of functions or basic blocks. The implementation of this phase of the process is covered in Section 5.1.3.

5.1.1 Tracer Engine Interface

Every new Tracer engine has to implement the abstract class `CollectEngine` and divide its collection process into phases represented by the abstract methods of this class. In order to use the engine from the command-line interface, it needs to be registered among the

supported engines in the definition of the `CollectEngine` class, and added as an option using the *Click*[1] library that is utilized to implement Perun's command-line interface.

When a user invokes the Perun collection process using a Tracer engine, the engine first creates the required temporary files and checks that the dependencies necessary for the successful execution of the collection process are satisfied. In this case, the files created are the pintool source file (`pintool.cpp`), its Makefile (`Makefile`) and a data file where the PIN output will be saved (this file has a unique name with each creation). The only hard dependencies of the collector are `pin` and `g++` (GNU C++ compiler) which need to be installed in the user's PATH (hence executable from the command-line). The installation of PIN might not be as straightforward, since the PIN framework officially provides only a development kit, which does not include an installation script.

Right after the necessary checks, there is a phase in which the engine is supposed to gather information about the available USDT probes, however, the PIN framework does not support the USDT probes. Therefore, the implementation of this method (`available_usdt`) returns an empty python dictionary every time it is executed, which effectively skips the phase entirely.

One of the key steps before the data collection itself begins is the creation of the pintool that will be used in the process. This is facilitated in the engine's `assemble_collect_program` method where the engine assembles a new pintool based on the specified requirements by the user. The generation of the core parts of the pintool is done using *Jinja2*[2] templates that allow a high level of flexibility. This is possible thanks to the design of pintools, which can be split into major semantic parts and easily assembled, in this case, by Jinja2. Since pintools are written in C/C++, the engine needs to generate the appropriate source code and then compile it using the `g++` compiler that the engine requires for this purpose. The PIN framework's kit includes a Makefile written for general compilation of any pintool inside the kit's directory, however, when integrating it with Tracer's engine, some minor changes regarding the Makefile invocation had to be done so that the compilation would be possible from the Perun's temporary folder. The creation of pintools and Makefile, along with implementation details regarding this phase, are covered in more detail in Section 5.1.2.

After the pintool is assembled, the collection process is ready to start. The `pin` command is executed in the method `collect` with the created pintool and binary along with the arguments for its execution specified by the user through the Perun's command-line interface. In this phase, PIN instruments and executes the provided binary (either in JIT or Probe mode, which handles the execution in a different manner) and populates the temporary data file with collected data.

Parsing of the gathered data is done immediately after its collection in the `transform` method. The purpose of this phase is not only to convert the collected data into a unified format, but also to validate and pair the records created by PIN, since there are two parts from which the record consists — the entry point record and the exit point record for each function or basic block execution. These two records contain the information that unambiguously identifies the program location and the time stamp when the entry occurred. These two records need to be combined into one which holds the data that identifies them and also the duration between the two points. However, sometimes the PIN

[1] See https://click.palletsprojects.com/en/latest/.
[2] See https://jinja.palletsprojects.com/en/latest/.

output contains only an entry point to a certain function, which signals a failed record and needs to be filtered out. The Section 5.1.3 dives further into the details of parsing the collected data.

At the end of the whole process, the engine executes the `cleanup` method, which removes temporary files and concludes the whole Tracer execution process.

5.1.2 Pintool and Makefile

When designing a pintool, one needs to define a method of program instrumentation, what data should be collected, and the format in which the data will be produced. The method defines not only the mode of instrumentation, but also its granularity. This work requires the implementation of multiple pintools in order to provide only the specified data and to use a specific mode. Since the collection of data and its transformation requires a substantial amount of time, collecting all of the possible types of data every time would be unnecessary, and especially time consuming. Therefore, the Tracer engine creates a new pintool with each execution, which provides the needed flexibility of the pintool.

The generation of a pintool is handled by Jinja2 templates that allow easy modification of pintool contents using python. Thanks to the general design of pintools, their structure can be split into three major parts, each representing one template:

- **Analysis:** defines the analysis functions where the timestamps are recorded and the output format is defined.
- **Instrumentation:** filters unnecessary instrumentation points based on the instrumentation granularity and assigns the analysis functions to these points, while also providing the necessary data to these functions which is then included in the collection output.
- **Main:** represents the main function of the pintool, where the necessary callback registrations are done, and the SUT is executed in one of the available modes (JIT or Probe).

Analysis template defines two types of analysis functions that are very similar. Analysis functions produce records before and after the execution of any function. These records contain information about the instrumented function (its name and identification number), but also the timestamp and the information where this record was created (either before or after the instrumented function) so that the output provides the necessary information to determine which records need to be combined for the calculation of the run-time of a function. Although, this might be enough to satisfy **FR_SRF (Support for recursive functions)** the information will not be enough to satisfy the **FR_SMA (Support multi-threaded/process applications)**, and therefore neither the **FR_FRT (Function run-times)**. In order to satisfy these requirements, the analysis functions need to output the thread and process identifiers, and also access the output file safely using the Mutex locking API[3] provided by the PIN framework. The definition of this output format can be found in the Listing 5.1.

[3]See the section about multi-threaded applications from: https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html#MT.

```
1 Granularity;Location;TID;PID;TimeStamp;RoutineID;RoutineName
```

Listing 5.1: Format of a single line in a PIN output with information separated by a semicolon. The format contains `Granularity` to differentiate basic blocks and routines, `Location` defining wheter the record is located before or after the routine, a thread identifier (`TID`), a process identifier (`PID`), a `TimeStamp` containing the time when the record was taken, and the `RoutineID` along with the `RoutineName` that identify the function the record coresponds to.

The timestamp creation is handled by the `OS_Time` function provided by PinCRT [18] and produces timestamps in microseconds. The selection of a suitable method to get the time stamp was already researched in the [19], where the `OS_Time` function was selected as the most suitable because its output does not require any further conversions or processing, and at the same time provides equal or slightly better performance than the other available methods. Performance is a very important part of analysis functions since they are executed before and after every instrumented function execution. PIN can inline these functions at the place of their execution and effectively eliminate the need to call these functions entirely. However, the functions need to be simple and have no complex control flow structures. Even though the timestamps in microseconds using this function are applicable, the future work should focus on providing timestamps in nanoseconds while also respecting inlining restrictions to ensure that the output is more accurate.

Instrumentation template defines a function that is called every time a new routine is found and decides if it should be instrumented, and in what way. The decision of which routines should be instrumented is based mainly on the function `IMG_IsMainExecutable` provided by the PIN framework. This function filters routines that are not part of the executable provided by the user. Another routines that are filtered are dynamically and artificially[4] created routines. When using the probed mode, the routines are also filtered by `RTN_IsSafeForProbedInsertion` which ensures that the PIN can insert an analysis function call before or after the function. After deciding if the function needs to be instrumented, the insertion of the analysis function call takes palace. This is done by using the `RTN_InsertCall` (or `RTN_InsertCallProbed` when using the probed mode) function and passing information about the instrumented routine (its unmangled name and identifier) and the analysis function to it. This means that the `RTN_InsertCall` function is called twice, since the insertion must happen before and after the instrumented routine.

Main template defines the `main` function for the pintool. This part of the pintool defines the granularity of the instrumentation right after the necessary initialization. To instrument routines, PIN provides `RTN_AddInstrumentFunction` which allows the registration of an instrumentation function (defined in the instrumentation template). The engine could use either image granularity or routine granularity for the instrumentation of routines. The image granularity is more efficient because the instrumentation function is called only once for every image and the routines contained in the image are instrumented right away. Thus, when instrumenting on the image-level, there is less context switching between PIN and the SUT. However, when combining routine instrumentation with the

[4] An artificial routine is an routine introduced by PIN for internal management and does not represent an actual routine in the application. See https://software.intel.com/sites/landingpage/pintool/docs/98547/Pin/html/group__RTN.html.

extensions described in Section 5.2, PIN failed to collect any data whatsoever, therefore the engine uses routine instrumentation granularity for the time being. At the end of the `main` function, the instrumentation mode is specified: either `PIN_StartProgram` in JIT mode or `PIN_StartProgramProbed` in Probe mode. This aims to satisfy the **FR_JM (Support for JIT mode)** and **FR_PM (Support for Probe mode)**.

Every template mentioned before is included in a single template, which represents the whole pintool. This template contains the necessary include statements, global variables, the definition of pintools argument (through which the name of the output file is specified) and the `Fini` function, which is executed before the end of the program. This way, the **FR_PI (Pintool implementation)** is satisfied since the Jinja2 templates provide a way of creating multiple different pintools that can be easily extended with additional functionality.

After the creation of a pintool source code, the Makefile for its compilation is generated similarly using a Jinja2 template. The Makefile defines the path to PIN's root (path to the PIN kit) where the default rules for pintool compilation are located, and defines the default rule which creates a folder for pintool shared object, and then recursively calls the default rule for its compilation. This creates the `pintool.so` which is then used in the invocation of the `pin` command as a tool that specifies the instrumentation process. This Makefile structure relies on files directly from PIN kit, which allows use of different kit versions just by exporting the `PIN_ROOT` variable containing absolute path to the kit before using the engine. However, this approach will probably change in the future work.

5.1.3 Transforming PIN Output Into the Perun Profile

When the data collection is finished, the output (see Listing 5.2 for an example) must be parsed and the gathered information transformed into a Perun profile. This process requires conversion of the PIN output format (see Listing 5.1), specified in the pintool, to the internal representation of this data and storing it in the memory until the proper pair record is found. After that, the two records merge into a single record that contains additional information such as the function's run-time. This new record contains all the information that needs to be stored in the Perun profile, therefore after merging all the records, a complete Perun profile can be created.

For the purpose of storing a record obtained from PIN, the class `RawDataEntry` was created. It serves not only as a simple data structure, but also defines methods that help with matching records and calculating the time delta of two records. The record format from PIN output is designed to be as easy to parse as possible. The majority of the information is stored as integers, except the name of the routine, which is outputted as a string.

When parsing the PIN output, the engine reads the output file line by line, each line representing a record. These records are stored as `RawDataEntry` and if the record contains information about a routine entry point, it is put into a backlog represented by a python list. This backlog is then searched for a matching entry point whenever a new exit point record is found. Adding a new record to the backlog must be done from the beginning of the list instead of just appending them at the end since the search for the index of pair record starts at its beginning, and the pair matched needs to be the last execution of the routine. This helps to satisfy the **FR_SRF (Support for recursive functions)** because accounting for the order of executed routines, the recursive routine call record could

```
0;0;0;107664;1651054391330820;15;QuickSort
    0;0;0;107664;1651054391330895;3;operator new[]
    0;0;0;107664;1651054391334661;18;Partition
        0;0;0;107664;1651054391334836;17;Swap
        0;1;0;107664;1651054391334843;17;Swap
        ...
    0;1;0;107664;1651054391335763;18;Partition
    ...
    0;0;0;107664;1651054391336328;18;Partition
        0;0;0;107664;1651054391336330;17;Swap
        0;1;0;107664;1651054391336333;17;Swap
        0;0;0;107664;1651054391336336;17;Swap
        0;1;0;107664;1651054391336338;17;Swap
    0;1;0;107664;1651054391336341;18;Partition
    0;0;0;107664;1651054391336453;7;operator delete[]
0;1;0;107664;1651054391338574;15;QuickSort
```

Listing 5.2: An example of an output featuring `QuickSort` function execution. The records are reduced and indented for better readability.

be mismatched since the PIN output does not contain any special identifiers for routine calls, which would unambiguously distinguish each call and eliminate the need of keeping the backlog sorted. After finding a pair of records, both are merged into a single record that contains the run-time of the routine (the time delta of the time stamps from the two records) along with the shared information by both original records. This new record is represented by the `FunctionCallRecord` class which serves as a simple data structure and has a method that converts the data to a python dictionary, suitable for storing in the Perun profile as a resource.

Since the common engine interface function `transform`, which takes care of the data transformation, needs to be a python generator, the merged records are immediately yielded as resources by this function. A resource represents one merged record and its format needs to be a python dictionary with specific keys — the fields containing information about the record. Since the fields were designed for the purpose of storing the information about function run-times, there is no need to register new ones when implementing basic Tracer functionality. This satisfies the **FR_PG (Profile generation)** requirement for the basic functionality of the tracer.

5.2 Extending the PIN Engine

With the basic Tracer functionality implemented, the extensions featuring the new capabilities can be built. The new functionality strives to leverage the PIN framework features to great extent and provide useful information in return. The collection of function arguments provides the user with crucial information which connects function run-times to its argument values and allows for further analysis of the run-time dependence on the value of specific argument. Gathering information about the function arguments is one of the

new additions to the Tracer's PIN engine. However, for this purpose the engine has to utilize the DWARF debug information contained in the binary, which is required for the arguments collection. In order to collect function argument values, the types and positions of the arguments need to be known before the pintool creation, because the pintool requires definition of separate analysis functions for each routine that needs the argument collection. The arguments collection can be enabled in both JIT mode and Probe mode, and can also be combined with another new feature: the collection of basic block run-times. The basic block run-times can be, however, collected only in JIT mode due to a restriction of the Probe mode. The Probe mode does not allow greater granularity of the instrumentation than routine level. Despite the restrictions, the basic block run-times can help the user pinpoint the source of slowdown in their programs better. In the following Section 5.2.1, the functional requirement **FR_FAI (Function argument information)** is addressed in more detail, and Section 5.2.2 breaks down the **FR_BRT (Basic block run-times)** requirement.

5.2.1 Arguments Collection

Although PIN provides a way of accessing the values of instrumented function's arguments, the pintool writer needs to know the position and the type of the argument before creating a pintool. This means that the pintool needs to be designed specifically for every instrumented program whenever the argument collection is involved, taking into consideration the functions with arguments that need to be collected. In order to gather the necessary information about function arguments, the engine uses the DWARF debug information stored in the binary and forms an internal representation in form of a python dictionary. The dictionary is then used to aid in the process of pintool creation and indirectly improves the visual representation of the collected data.

Obtaining the information about function arguments could be done in a few different ways. Static analysis of source code is one of them, however, the tools designed for this purpose tend to be limiting when it comes to the analysis of C++ source code. One of the tools considered for this purpose was CastXML [14], which is a tool that creates an XML tree from a C-family source code. Pairing this tool with pygccxml [16] would provide necessary information about the declared functions. Although this approach would work, it would introduce a new dependency in form of a program that is still in development after succeeding the GCCXML [15]. It would also need to be installed by the user and then executed by the engine as a separate process. Hence, instead of approaching this problem through static analysis, PIN engine extracts the necessary data from the DWARF debug information using the pyelftools [2, 3] library. Analysis of a binary file could be done using GNU binary utilities such as `nm` or `readelf`, however, using these utilities requires their execution which produces output that contains unnecessary information and needs to be parsed into an internal representation. Thus, using pyelftools to extract only the needed information from the structured DWARF format and converting it straight to the internal representation is a simpler approach. The binary provided by the user needs to be compiled with the debug information included, however, using newer versions of the `gcc` compiler includes DWARF version 5, which is not fully supported by the pyelftools yet. For this reason, the engine requires the binary to be compiled with `gcc` 7.5 which by default uses the DWARF version 4. A compact comparison of the tools considered for collection of function arguments data is shown in Table 5.1

	pyelftools	nm	readelf	CastXML
Produces information about functions	✓	✓	✓	✓
Produces information about function arguments	✓	✓	✓	✓
Produces unmangled function names	✓	✓	✓	✗
Analysis of debug information	✓	✓	✓	✗
Does not require additional dependencies	✓	✓	✓	✗
Does not require additional postporcessing	✓	✗	✗	✗
Is not executed as a separate process	✓	✗	✗	✗

Table 5.1: A comparison of the considered tools for collection of information about function arguments with the ideal requirements for the collection process shown in the first column.

When extracting the necessary information from the DWARF format using pyelftools, every debug information entry (DIE) must be read one by one to find the relevant entries. In this case, relevant DIE is distinguished by the tag `DW_TAG_subprogram` which contains the name of the function as one of its attributes (`DW_AT_name`) and has children DIEs that hold information about the argument parameters that can be distinguished by the tag `DW_TAG_formal_parameter`. The parameter name and type can be extracted from the parameter DIE. The name extraction is the same as the name extraction from a subprogram DIE but when it comes to the parameter type extraction, there is a set of predefined DIEs that define each part of the type, and together form its complete definition starts with the first type DIE that's referred to by a parameter DIE in its `DW_AT_type` attribute. Since this work focuses on basic types only, extracting them requires a recursive function that reads the chain of DIEs and forms the type definition (e.g. `unsigned long long int`). The supported types are integers `int` (including the extended versions with `long` or `unsigned`), character `char`, string `char*`, and boolean `bool`. The engine also supports collection of `double` and `float`, however, the values collected by PIN can be wrong, for example whenever a `float` or `double` argument comes after a pointer argument, such as `char*`. In these cases, the collected value does not match the expected value.

The extraction of information about function arguments from the binary is part of the engine's `assemble_collect_program` method which analyzes the binary using pyelftools whenever the user specifies that the collection of arguments should be included. The information provided by the pyelftools is then passed in form of a python dictionary to Jinja2 templates to be utilized in the process of pintool creation. The templates contain additional parts that were not present in the basic Tracer functionality.

The *analysis template* creates an additional analysis function for every routine that requires the arguments collection. These analysis functions also specify the set of arguments they collect (with the types provided by pyelftools) from the routine they belong to. This specially designed analysis function will be called before every execution of the instrumented routine it corresponds to, and its output format is extended by the values of the arguments of the instrumented function as shown in Listing 5.3.

The *instrumentation template* contains new logic that determines if the current routine has its own analysis function defined based on the name of the routine. For this purpose, the pintool template contains a global array of function names that have their own analysis functions. If the current routine's name is in the array, `RTN_InsertCall` contains the `IARG_FUNCARG_ENTRYPOINT_REFERENCE` along with the argument's index for each argument

```
Granularity;Location;TID;PID;TimeStamp;RoutineID;RoutineName;arg1;...;argN
```

Listing 5.3: The format of a single line in a PIN output when arguments collection is enabled. The format builds on top of the format defined in Listing 5.1 and adds the collected arguments to the end of the format.

```
0;0;0;162446;1651059266874581;19;QuickSort;10
        0;0;0;162446;1651059266874651;3;operator new[]
        0;0;0;162446;1651059266878176;22;Partition;0;9
                0;0;0;162446;1651059266878379;21;Swap
                0;1;0;162446;1651059266878385;21;Swap
                ...
        0;1;0;162446;1651059266879318;22;Partition
        0;0;0;162446;1651059266879618;22;Partition;5;9
                ...
        0;1;0;162446;1651059266879645;22;Partition
        ...
        0;0;0;162446;1651059266879851;22;Partition;2;3
                ...
        0;1;0;162446;1651059266879863;22;Partition
        0;0;0;162446;1651059266880047;8;operator delete[]
0;1;0;162446;1651059266882235;19;QuickSort
```

Listing 5.4: An example of an output featuring `QuickSort` function execution including the collected arguments of the functions executed in the process. The records are reducted and indented for better readability.

that should be collected, which signals PIN to pass the argument at the specified index as a reference. On the other hand, when the current routine's name is not specified in the array, the instrumentation uses the standard analysis function without arguments.

The main downside of this approach is the definition of analysis function for every routine that requires argument collection. These functions could be transformed into a single function using variadic arguments[5], however, this approach introduces unnecessary control flow logic to the analysis function which prevents applying the inlining mechanism provided by PIN for simple analysis functions. Even though using variadic arguments results in a compact and a more readable pintool, the additional overhead caused by non-inlined analysis functions is a bigger concern for this work.

Since the format of the PIN output (see Listing 5.4) is extended whenever the arguments collection is involved, the parsing of output needs to support this extension. Along with the format support, the argument values that are not numerical need to be converted into information that represents them better from the performance impact standpoint. For the basic arguments supported by the engine, the string `char*` is stored as the length of the string instead of the string itself, and the character `char` is stored as its Unicode value.

[5] Allows a function to accept any number of extra arguments. See https://en.cppreference.com/w/cpp/language/variadic_arguments.

```
1 Granularity;Location;TID;PID;TimeStamp;RoutineID;RoutineName;BasicBlockID
```

Listing 5.5: The format of a line representing a basic block record in a PIN output. The format builds on top of the format defined in Listing 5.1 and adds a basic block identifier to the end of the format.

To store the additional new information in a Perun profile, new keys for the resource dictionaries need to be defined in the `Profile` class. The keys can be *collectible* or *persistent* which defines their final form in Perun profile. The values of collectible keys are squashed into a single list for each resource that represents all of the executions in one place. The persistent values, on the other hand, are same for every execution and, therefore do not have to be stored multiple times. The keys defined for the arguments include a variable part in form of an index which on its own makes it impossible to define every possible key in an array. Thus, the keys are matched using a regex when translating the provided resources into a profile format.

5.2.2 Basic Block Run-times

One of the features provided by the PIN framework, and utilized in this work, is the instrumentation of basic blocks. The engine needs to create a suitable pintool for this purpose and be able to combine this feature with the function run-times collection and the arguments collection in the JIT mode, since the Probe mode does not support this feature. The pintool outputs additional information about the instrumented basic block, therefore the parsing of the output needs to be adjusted to this new format addition, and similarly the Perun profile also needs to contain this information. When the collection of basic blocks is involved, the size of the collected data naturally increases along with the time it takes to process this data.

This extension of functionality requires some additions to the created templates in the pintool creation process, while keeping in mind that the basic block collection parts of the pintool can not be generated if the Probe mode was selected. The *Analysis template* defines two new analysis functions dedicated to basic block data collection. The PIN output format (See Listing 5.5) for the basic blocks collection adds a way of deterministic identification of the basic blocks. The PIN framework API does not provide any form of internal identification for the basic blocks, so the address of the first instruction of a basic block was chosen as the identifier. This is possible thanks to the transparency of the PIN framework. Another part of the output added for the purpose of basic block run-times collection, is a granularity flag that differentiates a routine record from a basic block record.

The *Instrumentation template* defines a new function `Trace` which is called every time a new trace is found, and the *Main template* registers it as a callback whenever the collection of basic blocks is enabled using the `TRACE_AddInstrumentFunction` function. This granularity of instrumentation is utilized because the trace can be broken down into individual basic blocks that can be instrumented, as opposed to the routine granularity where each routine can be broken down into instructions it contains. The individual traces refer to the routine they are associated with, which helps to filter the traces and also contributes to the information outputted about the basic block. The filtering of traces based on routine and

```
0;0;0;169925;1651059918274929;22;Partition
       1;0;0;169925;1651059918274932;22;Partition;4197398
       1;1;0;169925;1651059918274934;22;Partition;4197398
       0;0;0;169925;1651059918274937;21;Swap
              1;0;0;169925;1651059918274940;21;Swap;4197353
       0;1;0;169925;1651059918274942;21;Swap
              1;1;0;169925;1651059918274945;21;Swap;4197353
       ...
       1;0;0;169925;1651059918274988;22;Partition;4197622
       1;1;0;169925;1651059918274991;22;Partition;4197622
       0;0;0;169925;1651059918274993;21;Swap
              1;0;0;169925;1651059918274996;21;Swap;4197353
       0;1;0;169925;1651059918274999;21;Swap
              1;1;0;169925;1651059918275001;21;Swap;4197353
       1;0;0;169925;1651059918275004;22;Partition;4197677
0;1;0;169925;1651059918275007;22;Partition
       1;1;0;169925;1651059918275009;22;Partition;4197677
```

Listing 5.6: An example of a PIN output for a Partition function when basic block collection is enabled. The records are reducted and indented for better readability.

image associated to it is similar to the `Routine` function intended for the function run-times collection. The `BBL_InsertCall` PIN API function could be used to instrument the basic blocks inside the trace, however, this approach proved to be unreliable since many basic blocks would not contain the exit point time stamp. This is due to the fact that in order to to use the `IPOINT_AFTER` with basic blocks, their last instructions needs to be a fall through instruction[6]. To ensure that the output data contains both entry and exit point time stamps of each basic block, the decision has been made to instrument only its first and last *instruction*, and for both insert the instrumentation code with `IPOINT_BEFORE` to avoid the PIN restriction. This is a reliable way of instrumenting every basic block, however, for the cost of loosing the execution time of the last instruction of a basic block. The problem of this approach will be addressed in the future work, by respectively assigning the additional time spent in a function to its basic blocks.

Transformation of the collected data (see example in Listing 5.6) into the Perun profile adapts to the new PIN output by creating an abstract class `Record` that represents a merged record, and having two separate implementations of this class which represent a function record (`FunctionCallRecord`) and a basic block record (`BasicBlockRecord`). When parsing the PIN output, records are distinguished by the granularity flag and stored in separate backlogs. This addition of a new backlog separates the two different record types based on their granularity, and ensures that the match is found sooner. The implementation of records matching is not much different from the version in Section 5.1.3, apart from the addition of separate backlogs and introduction of two different formats for the PIN output record based on its granularity.

[6] A fall through instruction is an instruction that does not change the control flow of the program and executes the next instruction immediately after it. See https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INS__INSPECTION.html#ga7602edb17e52e209492bab2c65fc1612.

5.3 Visualizations

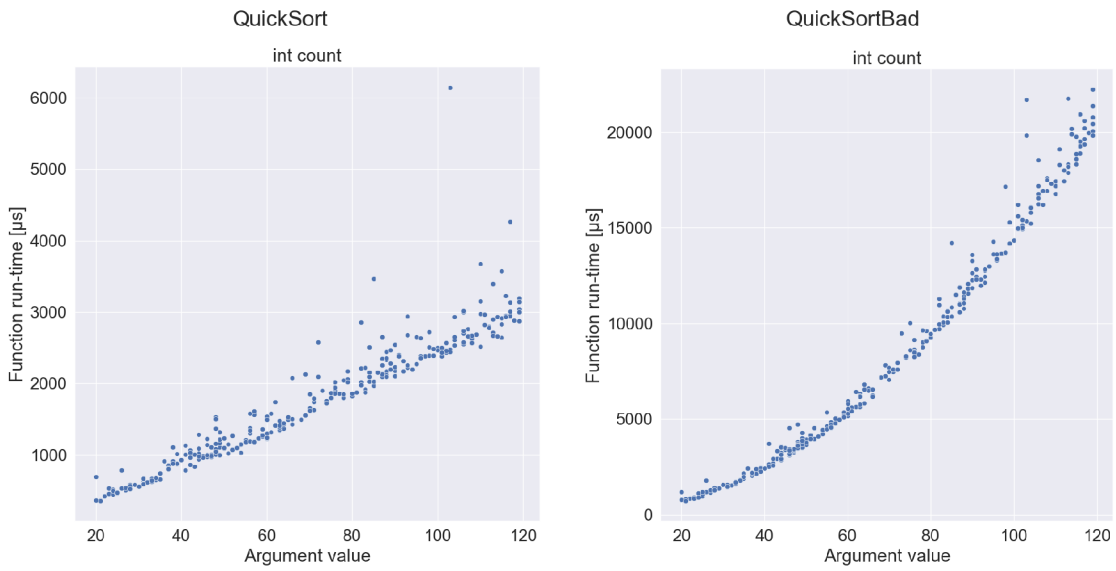
One of the essential parts of this work is the visualization of the data collected by the PIN engine. The manual analysis of the collected data is much easier when the data properties are highlighted with a specialized visualization that emphasizes them. This work focuses on visualization of the data collected by the new features introduced in the Tracer engine based on the PIN framework. Thus, two new visualizations are created in this work where one shows the dependence of function's run-time on values of its arguments. The second visualization contains a graph that illustrates the time spent exclusively in a function and on top of that, the time spent in the most time expensive basic blocks of this function. The other graphs created by this visualization is very similar, but shows the number of function executions instead of time.

Both of the visualization implementations process a Perun profile and output a visual representation of it. Before the visual representation is created, the Perun profile needs to be converted into a Pandas [22] `DataFrame` that contains relevant data for the visualization process. The visualization process is then realized either by utilizing the Seaborn [30] library in combination with the Matplotlib [13] library, or utilizing the Bokeh [4] library. The user can also influence the outcome of the visualization process as well as the data transformation by using Click command line options provided for given visualization.

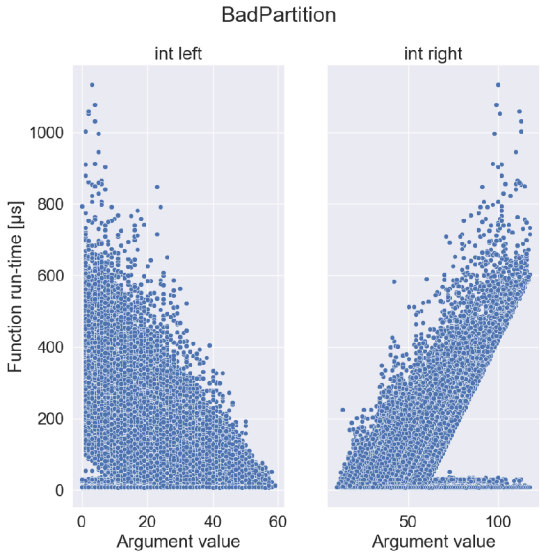
The function arguments visualization highlights the importance of their collection by showcasing the dependence of function run-time on them. This way, it aims to help the user identify the possible cause of slowdowns in a function. Using the Seaborn and Matplotlib libraries, the visualization utilizes the scatter plot (see Figure 5.1a), allowing the user to choose a function that has any arguments collected in the selected Perun profile, and to choose whether the arguments should be shown in a single graph as in Figure 5.1c, or individually as in Figure 5.1b. In either case, the graphs provide information about the arguments including their type and name.

The second visualization leverages the collected basic blocks data and essentially shows the impact of particular basic blocks on the run-time of a function. The visualization is designed to point out the most time demanding functions and provide a way of identifying the basic blocks that might be the cause of a slowdown. The filtering and extensive restructuring of the converted `DataFrame` has to take place to prepare the data for the visualization in a suitable format. The visualization process utilizes the Bokeh library, however, it respects the Perun restriction which limits the Bokeh version 0.12.6, instead of the most recent version. The graph created using this library is inspired by one of the examples in the gallery featured on the Bokeh website[7] that illustrates enhanced version of a sunburst graph. This graph (see Figure 5.2a) is created from individual annular wedges which represent either functions (parts of the background circle) or basic blocks (column-like shapes in the foreground of the circle). To control what is displayed in the graph, as presented in the Figure 5.2b, the user can filter by the most time consuming or the most called functions, and also limit the number of the most time consuming or the most called basic blocks. This visualization doesn't directly inform the user about the location of the basic block in their code. The future work will include this information, which requires deeper analysis of the PIN output, while taking into account that the basic blocks are discovered dynamically.

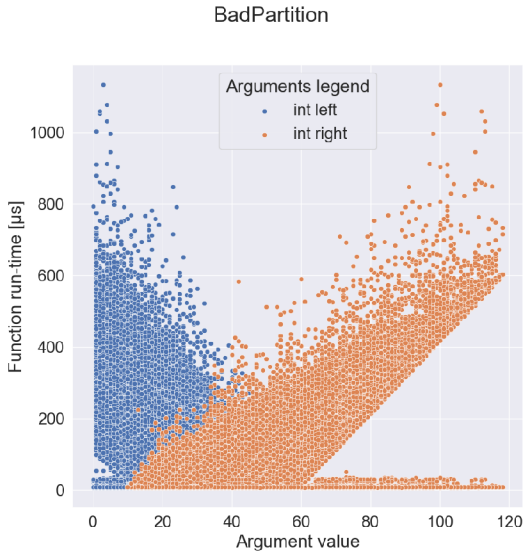
[7] See http://docs.bokeh.org/en/0.12.6/docs/gallery.html.



(a) Multiple functions featuring their argument values.

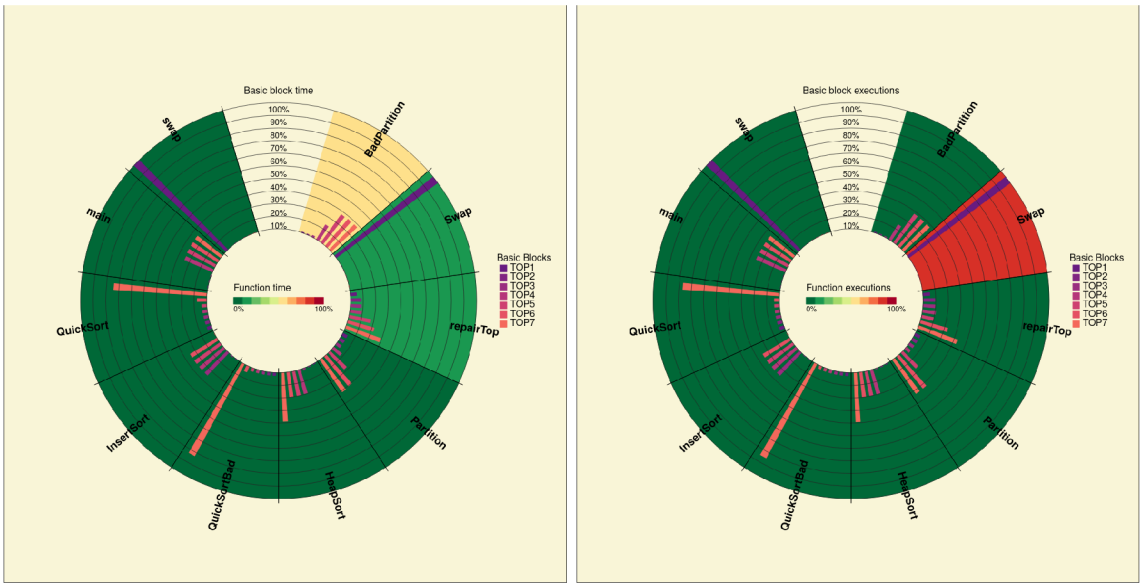


(b) A function with multiple arguments spread into multiple individual graphs.

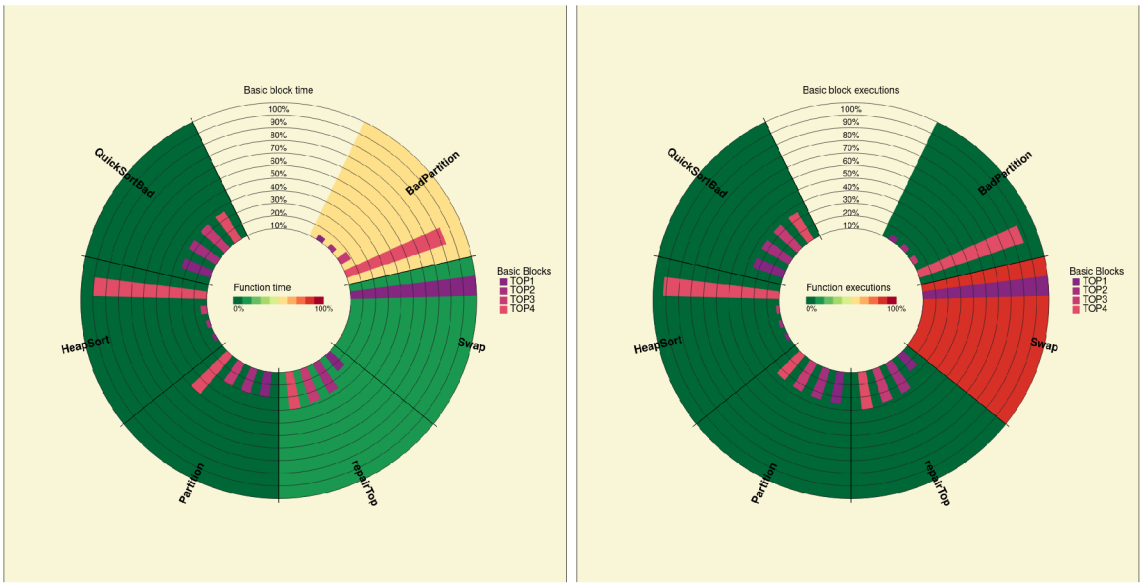


(c) A function with multiple arguments in a single graph.

Figure 5.1: An example of the arguments vizualization showing the dependence of function run-time on its argument values.



(a) All of the functions with some of the most expensive basic blocks.



(b) Top 6 functions filtered by their total exclusive time, including their top 4 basic blocks.

Figure 5.2: An example of the visualization of collected data regarding functions and their individual basic blocks, featuring graphs with the run-time on the left and graphs with the number of executions on the right. The large version of these images can be found in Appendix B.

Chapter 6

Experimental Evaluation

This chapter contains practical evaluation of the resulting Tracer engine, while verifying its functionality and outlining its beneficial impact. The description of the first experiment conducted on the new engine is provided in Section 6.1, demonstrating the benefits of collecting additional data alongside function run-times. For the purpose of this experiment, a program that implements multiple sorting algorithms was chosen to show the dependence of its performance on collected function arguments. The output of this experiment contains visual representation of the collected data created with visualization methods introduced in Section 5.3.

The second experiment described in Section 6.2 compares the new Tracer PIN engine to its other realizations which leverage the SystemTap and eBPF instrumentation frameworks. This experiment was conducted on a larger-scale project — `CCSDS` [1] compression program that is suitable for this purpose thanks to its wide range of functions. The output of this experiment is a comparison of the measured metrics of the execution of each engine, the size of the profiles created and the functions flagged as the most time consuming by each engine.

Machine specification. The experiments were conducted on a reference machine with the following specification:

OS	Fedora 34
Kernel	5.7.17
Arch	x86_64
CPU	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
RAM	32GB DDR3 @ 1600MHz
SSD	500GB Samsung SSD 840 @ 6Gb/s

6.1 Case Study #1: Impact of Increased Granularity

Evaluation of the new Tracer engine functionality, introduced in this thesis, strives to prove the benefits of obtaining additional information regarding the SUT to put the collected data into better context, and ultimately improve its analysis. This experiment also verifies the functionality of the data collection process utilizing both the arguments and basic block collection. The program used for this purpose implements multiple sorting algorithms (e.g.

Heap sort or Quick sort), along with an *improper* implementation of a Quick sort to show how beneficial the arguments and basic blocks collection is for identification of an improper implementation.

Methodology. The experiments conducted for this evaluation of the new engine utilize its collection of arguments and basic blocks at the same time. The collection was executed five times, where first two executions act as a warm up phase — to ensure the stability of the experiments, and therefore are excluded from the selection of the resulting data. The resulting data were selected from the remaining three executions based on median of the total execution time. The program used in this experiment executes different sorting algorithms in sequence 300 times while creating a reverse sorted array with random length from 20 to 200 elements for each of the sorting algorithms.

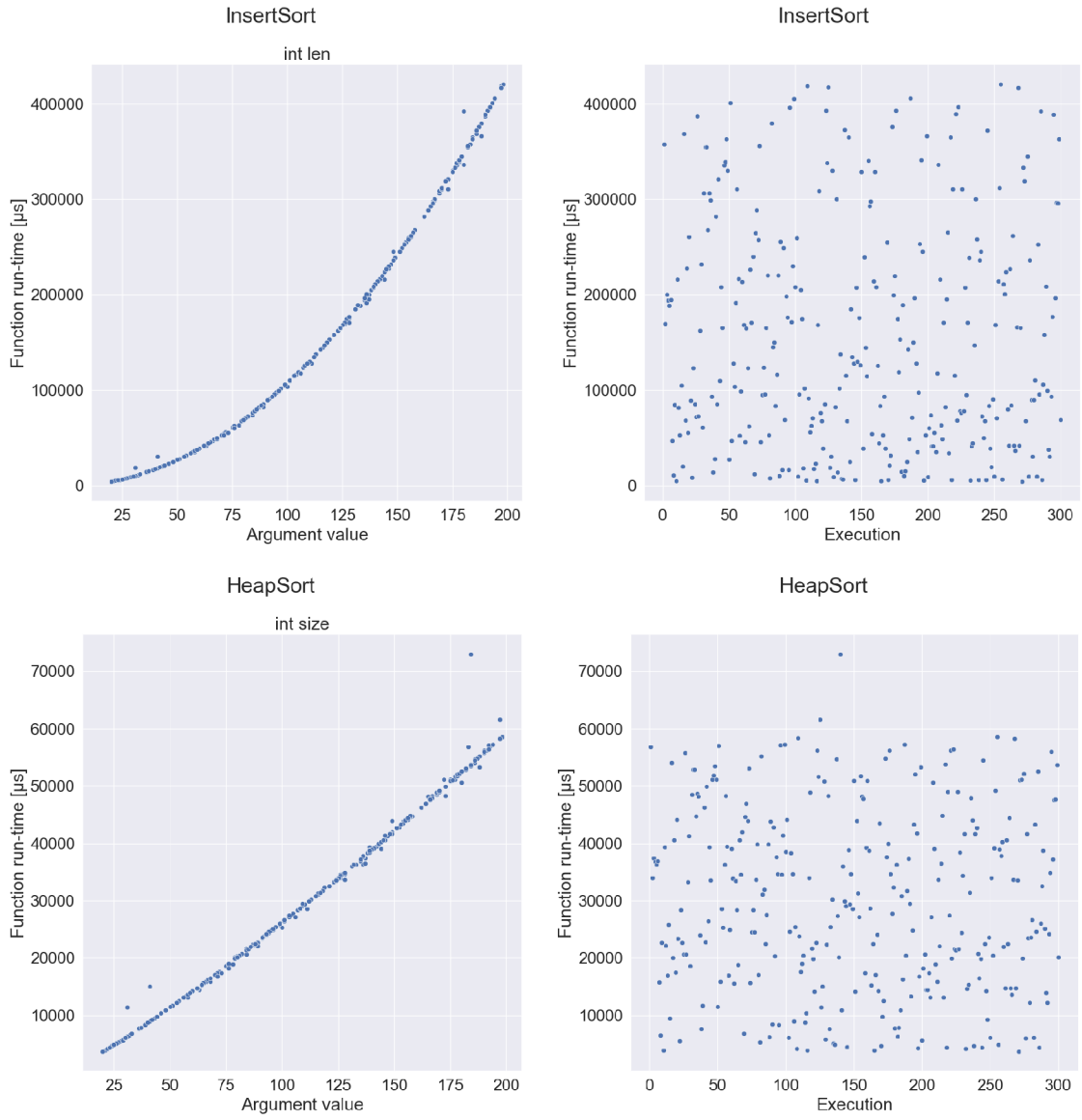


Figure 6.1: A visual comparison of having the values of function arguments available in addition to function run-times, and having just the run-times alone.

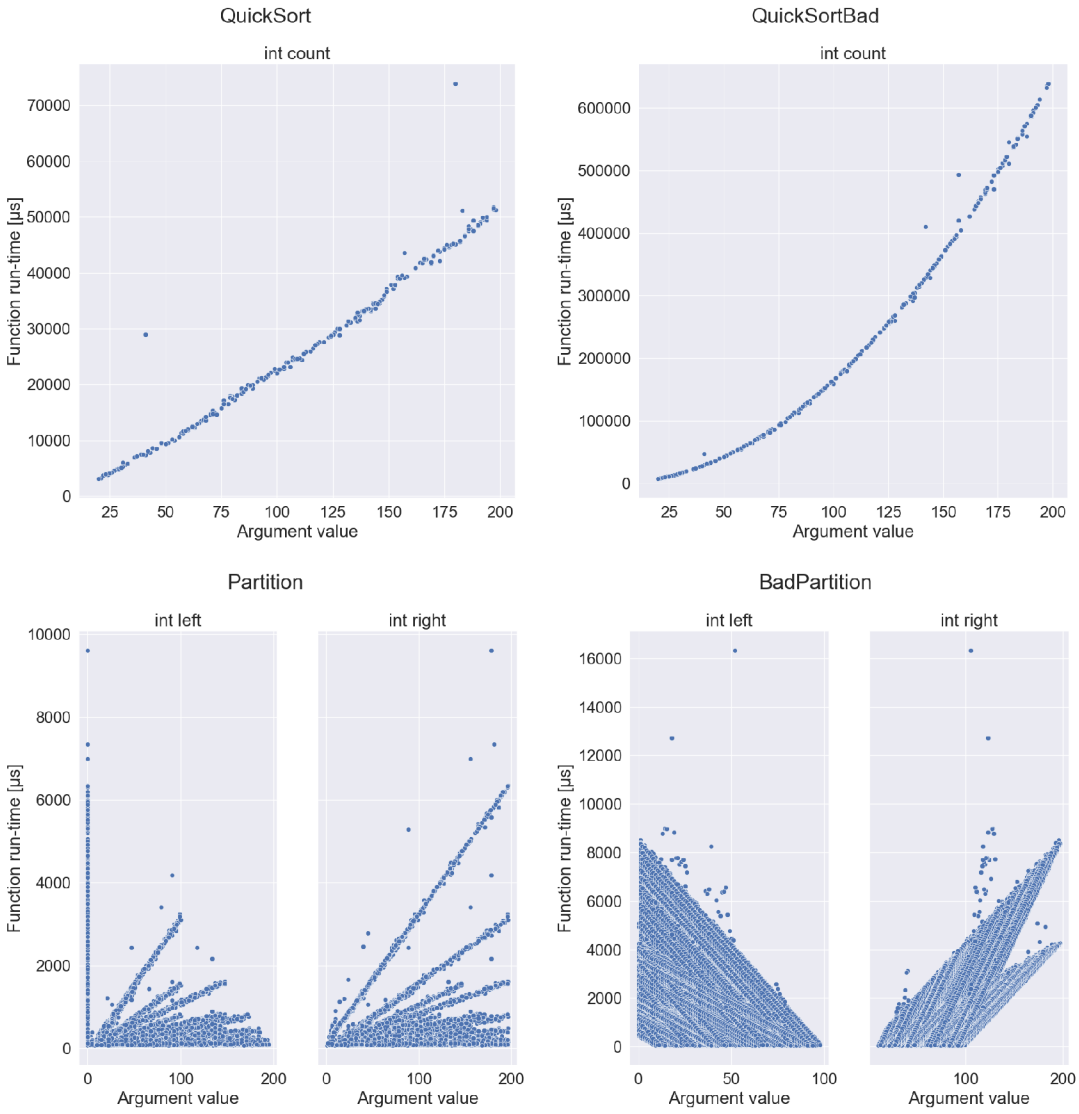


Figure 6.2: A visual comparison of two implementations of Quick sort and an auxiliary partitioning function used in the implementations. The graphs show increased complexity in the improper implementation of Quick sort (`QuickSortBad`).

The dependence between an argument value and a run-time of a function is an information obtained thanks to the collection of argument values shown in Figure 6.1 which presents the impact of the arguments collection in this experiment. The additional information puts significant amount of context to the collected data and shows the source of complexity of a function. When comparing the collected data of the improper Quick sort implementation to its correct implementation in Figure 6.2, the increased complexity of the improper implementation is easily visible.

Furthermore, the data collected regarding the basic blocks allows for analysis of time spent exclusively in each function, because the basic block run-times provide more detail about functions execution. This information proves to be beneficial when comparing the run-times of functions in Figure 6.3 where the `BadPartition` (a part of the improper implementation

of a Quick sort) stands out as the most time consuming function. When compared to its correct implementation (`Partition` function), the data shows significant differences.

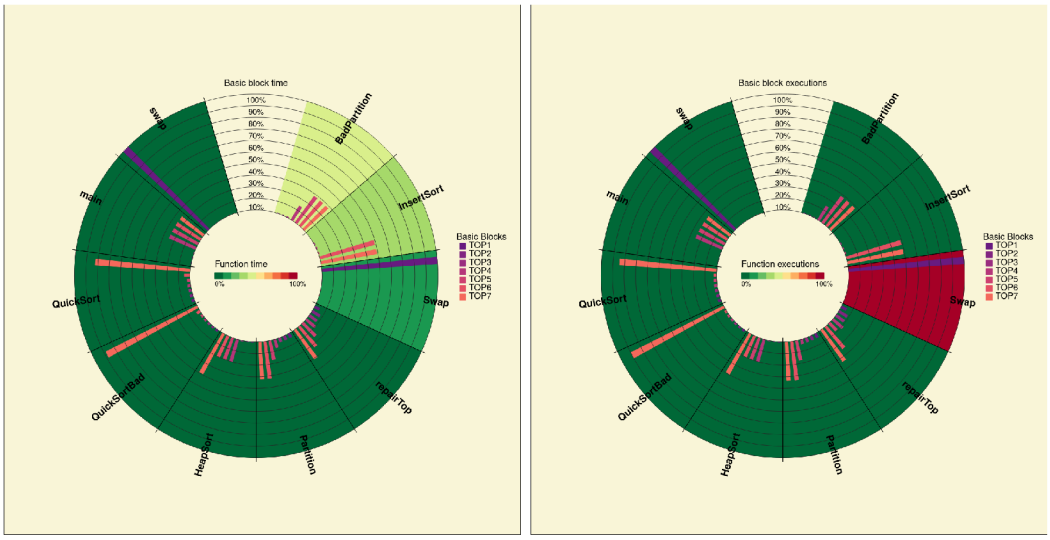


Figure 6.3: A visual interpretation of the collected basic block data, showing that the exclusive time spent in `BadPartition` function is significantly higher than any other function, indicating the source of the performance issue. The large version of these images can be found in Appendix B.

6.2 Case Study #2: Impact of Tracer Engines

The main goal of this experiment is to compare the performance of the newly implemented Tracer engine that leverages the PIN framework to the existing implementations that leverage eBPF and SystemTap instrumentation frameworks. The JIT mode was chosen to represent the PIN-based engine since the Probe mode currently does not reliably support the basic functionality this experiment strives to compare. Moreover, this experiment verifies the basic functionality of the resulting Tracer engine. The `CCSDS` compression program was chosen for this experiment as a larger-scale program so that the comparison of engines provides results comparable to real-world scenarios.

Methodology. The experiment conducted on the `CCSDS` compression program shares the methodology foundation with the first experiment. For every engine realization, the `CCSDS` program is executed five times (including the warm-up phase consisting of two executions) for three input images with different sizes. The Perun produces a set of metrics for the comparison, and a performance profile for each execution (excluding warm-up phase executions). This output is then analyzed, and based on the median of time spent in the engine, the representing execution is chosen and its metrics are used to compare the engines.

When comparing the time metrics obtained in the experiment, the smallest input image, presented in Table 6.1, shows that the PIN engine introduces significant instrumentation performance improvement for smaller inputs, while providing the same results as other engine implementations. However, further experimenting with larger `CCSDS` inputs shows that the new engine's instrumentation performs on par (See Table 6.2) or worse (See Table 6.3)

with the increased input complexity. This experiment also shows big differences between *Engine time* and *Instrumentation time* for the engine based on the PIN framework when compared to other engines. Both eBPF and SystemTap engines, however, feature optimizations of the output processing (such as parallel processing), whereas PIN engine does not feature optimizations yet, which is a part of the future work for this engine.

The *Profile size* shows differences among the engines, although the profile is in a unified format, the information stored in this format may vary from engine to engine. For example, SystemTap engine stores call-order for each function and also exclusive time spent in a function which, compared to eBPF-based engine and the new PIN-based engine, increases the profile size size significantly. The size might also be influenced by functions detected and instrumented by each engine. If one of the engine instruments a function that is called substantial number of times, and the other engines fail to instrument this function, the size of the resulting performance profile might increase for such engine. Every engine in this experiment instrumented nearly the same amount of functions while flagging the same function as the most time consuming, even though there are some differences among the top three most time consuming functions.

metrics \ engine	PIN	eBPF	SystemTap
Total time	**12.03s**	13.58s	13.38s
Engine time	**9.10s**	10.75s	10.57s
Instrumentation time	**1.59s**	9.20s	4.55s
Profile size	3.4MB	**1.4MB**	4.4MB
Instrumented functions	68	64	**72**

Top 3 func. names	`bpe_encode`
	`bpe_push_block`
	`bpe_encode_segment_bit_plane_coding`
		`bpe_encode`
		`bpe_encode_segment`
		`bpe_push_block`
			`bpe_encode`
			`bpe_push_block`
			`bpe_encode_segment`

Top 3 func. times	56.00% 55.46% 39.71%	60.28% 58.77% 39.30%	70.69% 70.51% 70.45%

Table 6.1: A comparison of the Tracer engines using the `CCSDS` compression program with an image size of 112x112 pixels. The *Total time* represents the time of the whole process from the start to the end of Perun execution. The *Engine time* represents a portion of the Total time taken by the given engine, and the *Instrumentation time* corresponds to the instrumentation part of the engine's execution.

Although, the implementation of PIN's Probe mode is described in the Chapter 5, its testing after the implementation and during this experiment uncovered inconsistency with instrumentation *after* a routine. It requires creation of a `PROTO`[1] object based on the *instrumented routine* which wasn't entirely clear before the implementation and had to be tested. However, even with proper information for creation of this prototype, PIN allows the `float` and `double` data types only as return types for the prototype, and doesn't support them as arguments. This makes Probe mode viable only for certain functions that meet the specified requirements and therefore hard to use for general instrumentation of routines in Tracer engine.

[1]Prototype of a routine, see https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__PROTO.html.

metrics \ engine	PIN	eBPF	SystemTap
Total time	42.50s	**21.12s**	27.93s
Collector time	37.77s	**17.22s**	21.39s
Instrumentation time	**5.89s**	10.38s	7.03s
Profile size	17MB	**6.4MB**	24MB
Instrumented functions	68	66	**72**

Top 3 func. names	`bpe_encode`
	`bpe_push_block`
	`bpe_encode_segment_bit_plane_coding`
		`bpe_encode`
		`uint32_abs`
		`bpe_push_block`
			`bpe_encode`
			`bpe_push_block`
			`bpe_encode_segment`

Top 3 func. times	66.83% 66.55% 48.86%	62.60% 19.90% 19.46%	73.28% 73.09% 73.04%

Table 6.2: A comparison of the Tracer engines using the `CCSDS` compression program with an image size of 256x256 pixels. The *Total time*, *Engine time* and the *Instrumentation time* have the same meaning as described in Table 6.1.

metrics \ engine	PIN	eBPF	SystemTap
Total time	158.19s	**45.65s**	77.58s
Collector time	146.79s	**38.34s**	59.52s
Instrumentation time	21.05s	**14.70s**	15.51s
Profile size	66MB	**23MB**	93MB
Instrumented functions	68	69	**72**

Top 3 func. names	`bpe_encode`
	`bpe_push_block`
	`bpe_encode_segment_bit_plane_coding`
		`bpe_encode`
		`uint32_abs`
		`bpe_encode_segment`
			`bpe_encode`
			`bpe_push_block`
			`bpe_encode_segment`

Top 3 func. times	69.48% 69.26% 50.49%	63.65% 20.52% 15.66%	73.96% 73.77% 73.72%

Table 6.3: A comparison of the Tracer engines using the `CCSDS` compression program with an image size of 512x512 pixels. The *Total time*, *Engine time* and the *Instrumentation time* have the same meaning as described in Table 6.1.

Chapter 7

Conclusion

This thesis presents an extension of the Performance Version System – Perun in form of a new Tracer collector engine based on the PIN instrumentation framework. This newly implemented engine features basic functionality of the existing Tracer engines, and furthermore extends the granularity of the collected data by implementing function arguments collection and basic block run-times collection. The new information gathered by the PIN-based engine can be manually analyzed by the user thanks to the two new visualization techniques (based on the scatter plot and modified version of sunburst graph) introduced in this work. All of the functional requirements set prior to implementing the new engine were met, however, the utilization of the PIN's Probe mode has some significant restrictions which render it unstable.

The evaluation of the new engine features an experiment conducted on numerous sorting algorithms to show the positive impact of collecting values of function arguments and its interpretation with new visualizations. This experiment proved that the function arguments can help analyze the source of complexity of a function, and that the basic block run-times can help with locating the source of slowdown even further, while also estimating the exclusive time spent in each function. Second experiment, conducted on a larger-scale image compression program `CCSDS`, compares the basic functionality of the new Tracer engine to the other engines based on eBPF and SystemTap instrumentation frameworks. The PIN framework introduces significant performance improvement of instrumentation when smaller input sizes are used, and comparable performance with larger inputs. Moreover, each engine flagged the same functions as the most time consuming.

Future work. One of the main future goals will be the optimization of the PIN output transformation to the Perun profile, which is now a major source of performance issues in the new engine. The performance and memory usage could be further optimized by better routine instrumentation filtering, which would reduce the number of unwanted instrumented routines to minimum. Basic blocks collection could be improved by providing the user with an easy way of connecting a basic block to the source code, as well as by approximating the last instruction run-time, which could not be consistently instrumented due to the PIN restrictions. The future work will also extend the set of collectible argument types beyond the currently supported set of basic types.

Bibliography

[1] *Lossless Data Compression, Recommended Standard: CCSDS 121.0-B-3.* Washington, DC, USA: The Consultative Committee for Space Data Systems, August 2020.

[2] Bendersky, E. *Pyelftools* [online Github Repository]. [cit. 2022-04-23]. Available at: https://github.com/eliben/pyelftools.

[3] Bendersky, E. *User's Guide* [Online Github Repository Wiki]. [cit. 2022-04-23]. Available at: https://github.com/eliben/pyelftools/wiki/User's-guide.

[4] Bokeh Development Team. *Bokeh: Python library for interactive visualization.* 2018. Available at: https://bokeh.pydata.org/en/0.12.6/.

[5] Bruening, D. L. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation* [online]. USA, 2004. [cit. 2022-01-30]. Dissertation. Massachusetts Institute of Technology. Available at: https://dspace.mit.edu/handle/1721.1/30160.

[6] Calavera, D. and Fontana, L. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking.* O'Reilly Media, Incorporated, 2019. ISBN 9781492050209. Available at: https://books.google.sk/books?id=--tlwwEACAAJ.

[7] Cohn, R. *Pin Tutorial* [online]. 2009 [cit. 2022-05-06]. Presentation slides. Available at: https://www.intel.com/content/dam/develop/external/us/en/documents/pintutorial-academiasinica-1.ppt.

[8] Devor, T. *Pin: Intel's Dynamic Binary Instrumentation Engine* [online]. 2013 [cit. 2022-01-30]. Presentation slides. Available at: https://www.intel.com/content/dam/develop/external/us/en/documents/cgo2013-256675.pdf.

[9] Fiedor, T. *Perun: Lightweight Performance Version System* [online Github Repository]. [cit. 2021-12-18]. Available at: https://github.com/tfiedor/perun.

[10] Fiedor, T. and Pavela, J. *Perun Documentation: Release 0.20.2. 2021.* [online]. [cit. 2021-12-18]. Available at: https://github.com/tfiedor/perun/blob/master/docs/pdf/perun.pdf.

[11] Gregg, B. *Choosing a Linux Tracer (2015)* [online]. [cit. 2022-01-30]. Available at: https://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html.

[12] Gregg, B. *BPF Performance Tools: Linux System and Application Observability.* 1stth ed. Addison-Wesley Professional, 2019. ISBN 0136554822.

[13] Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*. IEEE COMPUTER SOC. 2007, vol. 9, no. 3, p. 90–95. DOI: 10.1109/MCSE.2007.55.

[14] Kitware. *CastXML* [online Github Repository]. [cit. 2022-04-23]. Available at: https://github.com/CastXML/CastXML.

[15] Kitware. *GccXML* [online]. [cit. 2022-04-23]. Available at: https://www.gccxml.org/.

[16] Kitware. *Pygccxml* [online Github Repository]. [cit. 2022-04-23]. Available at: https://github.com/CastXML/pygccxml.

[17] Krátký, R., Jahoda, M., Domingo, D. and Cohen, W. *SystemTap Beginners Guide* [online]. [cit. 2022-01-30]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/systemtap_beginners_guide/index.

[18] Levi, O. Pin - A Dynamic Binary Instrumentation Tool. *Intel* [online], 13. june 2012. 2021-10-28 [cit. 2022-04-24]. Available at: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[19] Liščinský, M. *Performance Analysis Based on Noise Injection*. 2020. Master's Thesis. Brno University of Technology, Faculty of Information Technology.

[20] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. Jun 2005, vol. 40, no. 6, p. 190–200, [cit. 2022-01-30]. DOI: 10.1145/1064978.1065034. ISSN 0362-1340. Available at: https://doi.org/10.1145/1064978.1065034.

[21] Lukan, D. *Pin: Dynamic Binary Instrumentation Framework* [online]. [cit. 2022-01-30]. Available at: https://resources.infosecinstitute.com/topic/pin-dynamic-binary-instrumentation-framework/.

[22] McKinney, W. et al. Data structures for statistical computing in python. In: Austin, TX. *Proceedings of the 9th Python in Science Conference*. 2010, vol. 445, p. 51–56.

[23] Merey, A. *What are BPF Maps and how are they used in stapbpf* [online]. [cit. 2022-01-30]. Available at: https://developers.redhat.com/blog/2017/12/15/bpf-maps-used-stapbpf#.

[24] Michael J. Eager, E. C. *Introduction to the DWARF Debugging Format* [online]. April, 2012 [cit. 2022-05-06]. Available at: https://dwarfstd.org/doc/DWARF4.pdf.

[25] Nicholas Nethercote, J. S. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. [online]. 2007, [cit. 2022-01-30]. Available at: https://valgrind.org/docs/valgrind2007.pdf.

[26] Pavela, J. *Data structures profiling library for C/C++ programs*. 2017. Bachelor's Thesis. Brno University of Technology, Faculty of Information Technology.

[27] Pavela, J. *Efficient techniques for program performance analysis*. 2020. Master's Thesis. Brno University of Technology, Faculty of Information Technology.

[28] Srinivasaraghavan, R. Exploring the DWARF debug format information. [online]. august 2013, [cit. 2022-05-06]. Available at: https://valgrind.org/docs/valgrind2007.pdf.

[29] Stupinský Šimon. *New models for automatic detection of performance degradation*. 2019. Bachelor's Thesis. Brno University of Technology, Faculty of Information Technology.

[30] Waskom, M. L. Seaborn: statistical data visualization. *Journal of Open Source Software*. The Open Journal. 2021, vol. 6, no. 60, p. 3021. DOI: 10.21105/joss.03021. Available at: https://doi.org/10.21105/joss.03021.

Appendix A

Contents of the included storage media

```
/
├── perun/ ..... Perun implementation along with the PIN Tracer engine
├── pin-3.22 ..... PIN kit
├── example-programs/ ..... Programs for functionality testing
├── vm-files/ ..... Files for setup of virtual machine with perun pre-installed
├── README.md ..... Useful information about the storage medium content
├── thesis-source/ ..... LaTeX source code of this thesis
└── xmocar00-thesis.pdf/ ... Digital version of this thesis
```

Appendix B

Basic Block Visualization Examples

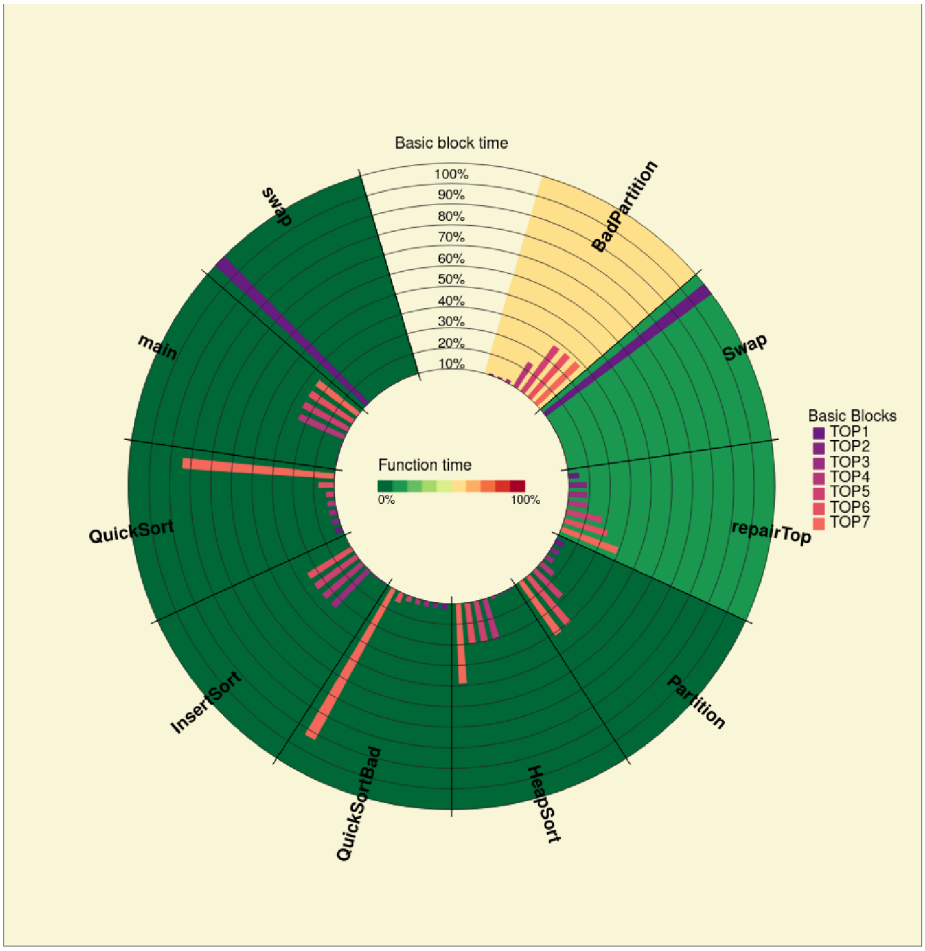


Figure B.1: An example of the visualization of collected data regarding functions and their individual basic blocks featuring all of the functions of the tested program and some of the most time expensive basic blocks time consumed.

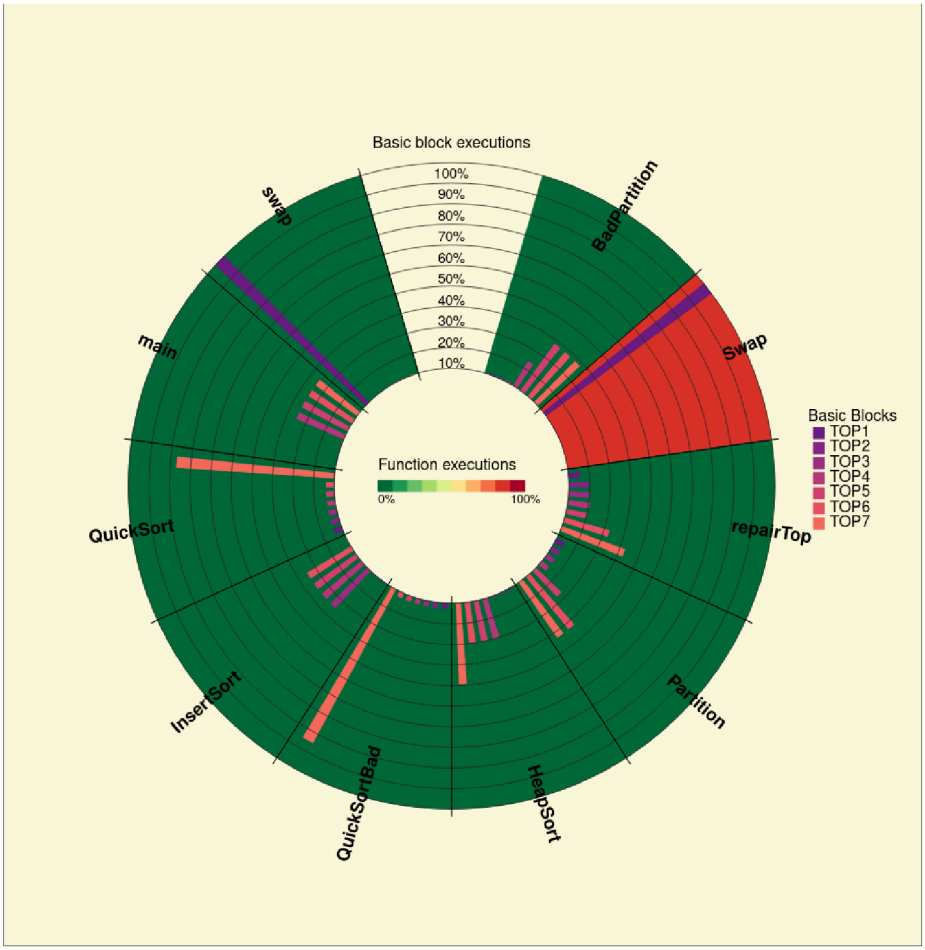


Figure B.2: An example of the visualization of collected data regarding functions and their individual basic blocks featuring all of the functions of the tested program and some of the most time expensive basic blocks and their executions. Paired with B.1

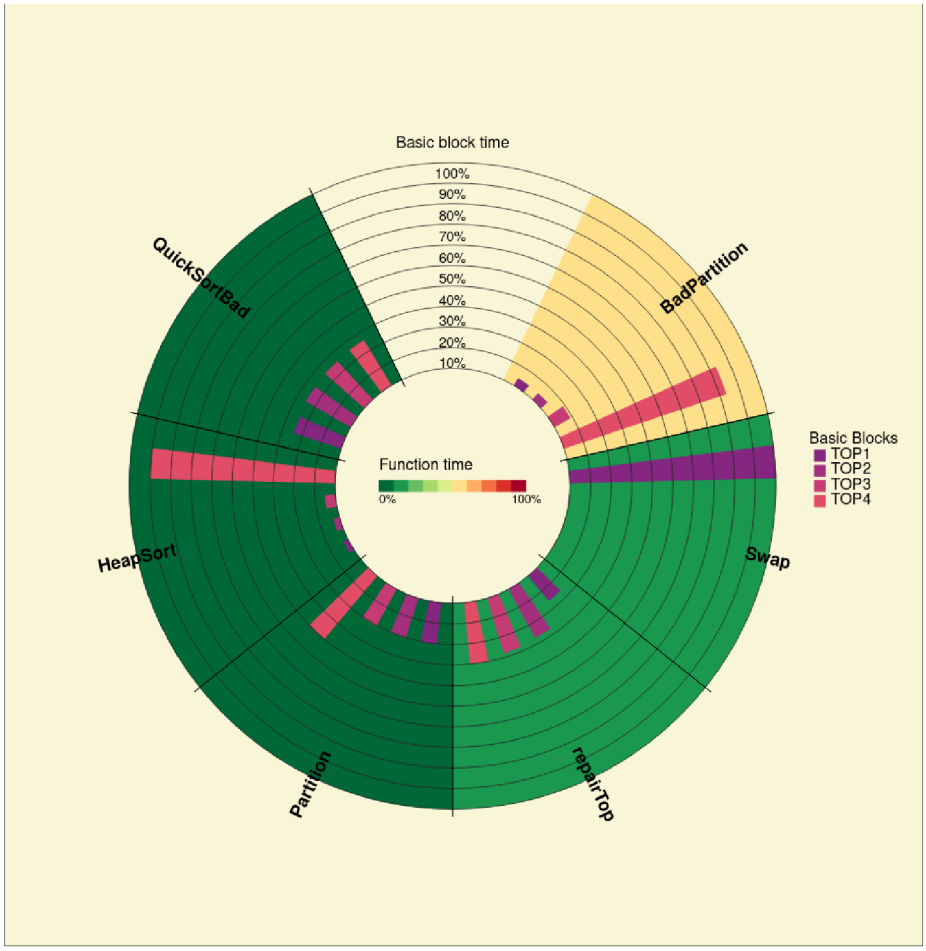


Figure B.3: An example of the visualization of collected data regarding functions and their individual basic blocks featuring top 6 most time consuming functions sorted by time including their top 4 basic blocks.

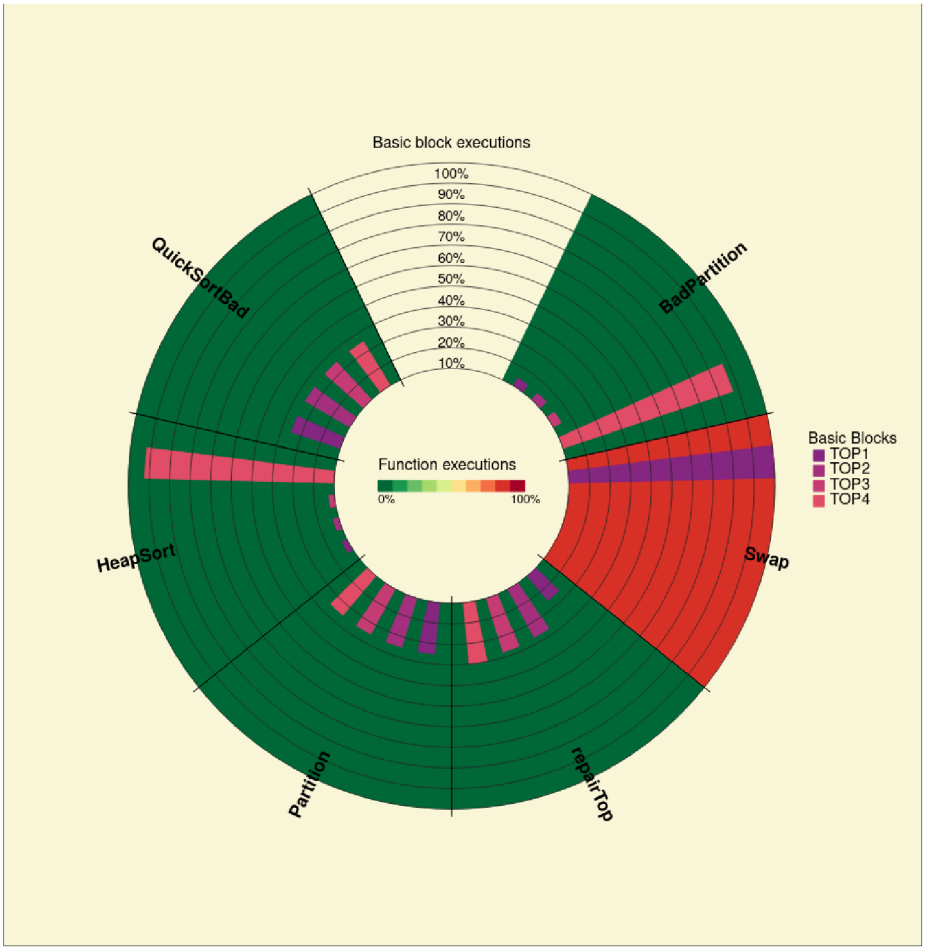


Figure B.4: An example of the visualization of collected data regarding functions and their individual basic blocks featuring top 6 most executed functions sorted by time including their top 4 basic blocks. Paired with B.3

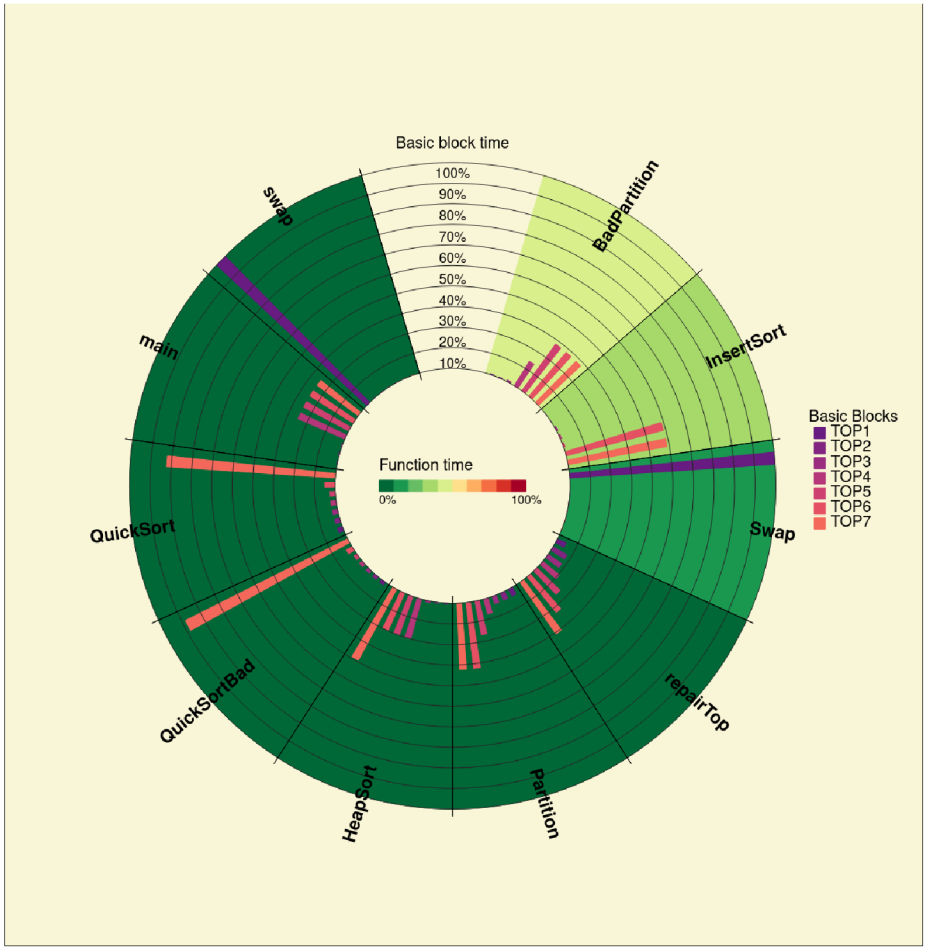


Figure B.5: A visual interpretation of the collected basic block data, showing that the exclusive time spent in `BadPartition` function is significantly higher than any other function indicating the source of the performance issue.

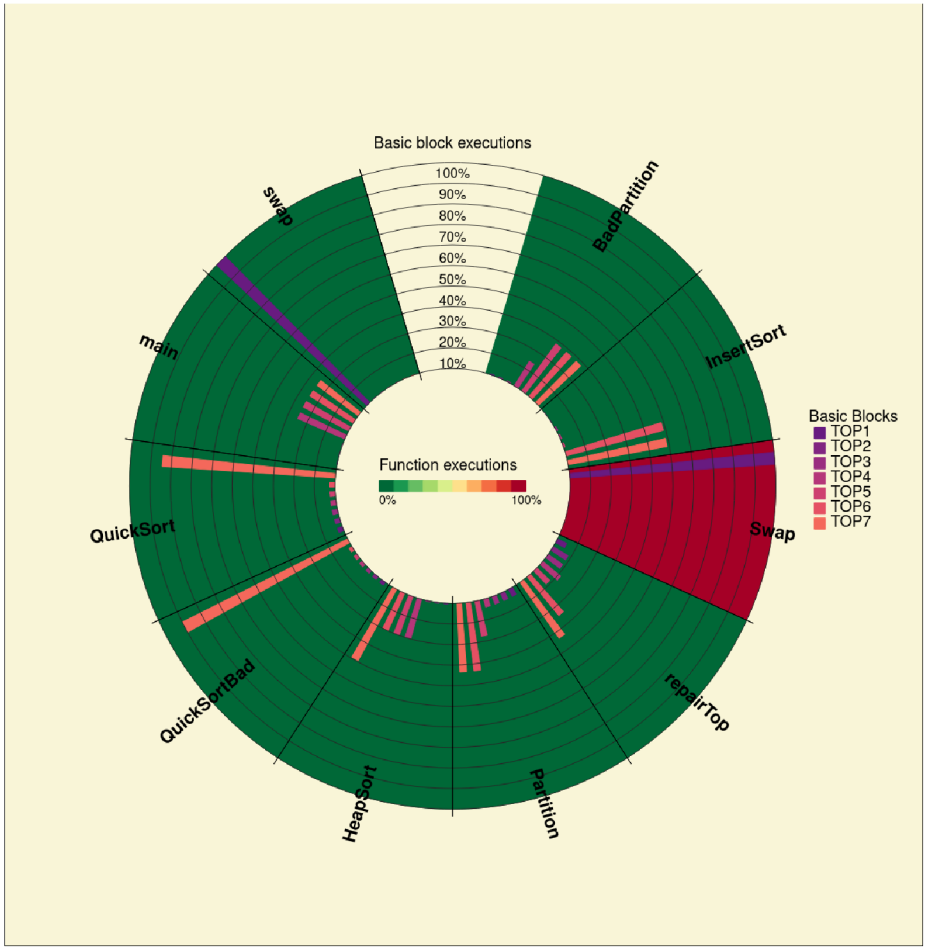


Figure B.6: A visual interpretation of the collected basic block data, showing the executions of functions. Paired with B.5