On More Effective Performance Testing

FOOBAR

Supervisor: Prof. Neeraj Suri
Dr. Matthew Bradbury

School of Computing and Communications
Lancaster University

This dissertation is submitted for the degree of
Doctor of Philosophy

System Security Group September 2023

This page is intentionally left blank

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

FOOBAR
September 2023

Acknowledgements

As a life long gamer since 90s, it seems no surprise that I walk on the path of computer science. It is not surprising after all that, besides games, I started with the LOGO language in the kindergarten, when I could hardly enumerate 26 letters across the keyboard. I still remember my first X86 computer, a Pentium 133 MHz laptop (people had really strong and high-temperature tolerant laps in 90s!) with old Windows operating systems (OS). The laptop has a spring loaded compact disc (CD) drive, whose clutch is broken and could eject while the computer is accessing data on the CD. When the CD is ejected unexpectedly, the system crashes with a blue screen claiming the accessed memory address is invalid. A hard reboot upon the blue screen sometimes loses the data pending at I/O cache queues, which eventually corrupts the hard drive data and the Windows graphic interface is no longer able to start properly. After 20 years, I know this is a system reliability problem and a security vulnerability in terms of system availability. More specifically, it is an *exaggerated error handling* problem, discussed by Pakki and Lu [122]. Perhaps, I am destined to research software and system reliability problems 20 years ago.

First and foremost, I would like to thank my PhD supervisor and mentor, Prof. Neeraj Suri, PhD. I appreciate the opportunity you granted me, and your constant support on this tortuous journey of computer science research. I am also grateful for your invaluable suggestions for my research. It is an honor to work under your advice, along with the great groups you built in Technische Universität Darmstadt and Lancaster University. Also I would like to thank Dr. Matthew Bradbury for your help and edits on my writings.

Thank you Stefan for your introduction into the fun world of computer science research, as well as your senpai experience not only in computer science, but also in craft beers. I would also like to thank Olli, for your help during my time in UK, and your guidance towards quality liqueurs and spirits. Thank you Salman and Heng for having to listen to my complains and non-sense chit-chat all the time from Germany to UK, physically and remotely. Thank you Nico for nitpicking my C++ code from time to time and that is a lot of fun to explore how ugly C++ actually is. Also a thank you to Habib for drinking together with Stefan, Olli and Nico. That was a seriously important part of a PhD life to keep sanity! Thank you Sabine, for your tremendous help in administrative work and free us from repetitive

and tedious documents in German. I also owe Ute a big thank you for your infrastructure administration, which is so crucial that saves us tons of time to focus more on our researches! I am also grateful for being together with all other former DEEDS group members, Ahmed, Hatem, Kubi, Patrick and Tsveti. It was such a memorable and amazing period to work with you! I am thankful for Prof. Roberto Natella as my co-author, as well as your support and suggestions backed by profound experience in computer science!

I would thank all members in the system security group at Lancaster University as well.

Abstract

Software on modern computer systems is ubiquitous in our daily lives, running on a wide range of devices, from smart watches to super computers. Such software systems grant us with the versatility of their functionality, and become increasingly complex during years of development to adapt the increasing demands of users. However, the software systems may not be dependable per se, on which we rely more and more in recent years, as the human mistakes in the code could turn to bugs, defects or vulnerabilities. To minimize both the number and the severity of such human mistakes, developers usually test the software before releasing it to public. Software tests verify if the behavior of the underlying software, or software under test (SUT), deviates from a specification, which states the required functional or non-functional properties of the SUT. One of the key aspects of non-functional properties is the software performance, which measures how fast a SUT processes its inputs. In comparison with functional testing, performance testing lacks testing oracles, is flaky with testing environment. In this thesis, a piece of inefficient code causing the SUT to deviate from specification is considered as a *performance bug*. Developers usually utilize performance testing similar to aforementioned functional testing to identify performance bugs. This thesis aims to improve performance testing by *performance mutation testing* (PMT) and *performance fuzzing*.

A natural question is, as developers err in the SUT, a test writer may also make mistakes when devising the test suite, and it remains unknown whether performance testing is calibrated correctly to be able to find performance bugs. So, an approach to grade the quality of performance testing is needed. As performance testing usually lacks testing oracles, the first step towards evaluating performance testing is to study and understand existing performance bugs. A complete study of more than 700 performance bugs with the discussion on their semantic commonalities is hence presented.

Based on investigated performance bugs as well as the extracted semantics, the *mutation testing* (MT) technique is adopted to grade the effectiveness of performance testing. MT injects *known bugs* into SUT, and checks if the testing suite is capable of identifying artificially injected bugs. In addition, MT could also be used to determine if injected bugs are covered, thus determining the code coverage of the testing suite. Performance bugs to be injected are

synthesized based on the fault models derived from the aforementioned investigation. In this thesis, an effective PMT framework is developed and evaluated.

PMT tells merely how effective performance testing is, and a lot of manual efforts are nevertheless needed to find proper inputs for a test case. In recent years, *fuzz testing* (or *fuzzing* for short), was utilized to search for proper inputs for SUT automatically. Despite existing approaches exploited the capability of fuzzing to identify performance bugs, the existing discussion mainly focuses on how a performance fuzzer could explore the longest execution path. An early experiment shows that each node of the execution path, on which fuzzing is guided, may not yield the same performance impacts. As many factors could impact the performance of generated inputs, e.g., fuzzing timeout limits, this thesis reevaluates a performance testing framework, known as PERFFUZZ, to provide general recommendations on using performance fuzzing.

In summary, this thesis a) provides a dataset of performance bugs, and b) presents a PMT framework accepting performance bug semantics, and c) investigates significant configurations of performance fuzzing and provides suggestions on the effective usage of performance fuzzing.

Table of contents

List of figures **xv**

List of tables **xvii**

1 Introduction **1**
1.1 Software Dependability and Security 1
1.2 Performance Testing and Performance Bugs 4
1.3 Research Questions and Contribution 6
1.4 Publications 10
1.5 Organization 11

2 Related Work and Background **13**
2.1 Performance Issues and Diagnostic Approaches 13
2.2 Code Instrumentation and Performance Profilers 15
2.3 Empirical Study on Performance Bugs 18
2.3.1 Performance Bugs for Evaluating and Training Detection and Localization Approaches 18
2.3.2 Empirical Studies of Performance Bugs 19
2.4 Performance Mutation Testing 19
2.5 Performance Fuzzing 22

3 Empirical Study on Performance Bugs **25**
3.1 Introduction 25
3.2 Methodology 27
3.2.1 Selection of Projects and Commits 27
3.2.2 Taxonomy 29
3.2.3 Bug Fix Time 30
3.2.4 Seniority of Fixers 31
3.2.5 Number of Changed Lines 32

3.2.6 Bug Collections . . . 33
3.3 The Shape and Variety of Fixed Performance Bugs . . . 33
3.3.1 Fast-path . . . 33
3.3.2 Arguments . . . 34
3.3.3 Cache memoization . . . 35
3.3.4 Data Access . . . 36
3.3.5 Synchronization . . . 38
3.3.6 Miscellaneous . . . 38
3.4 Performance Bugs Characteristics . . . 39
3.4.1 Bug Pattern Distribution . . . 39
3.4.2 Performance Bug Fix Duration . . . 41
3.4.3 Performance Bug Fixing Developer Experience . . . 43
3.4.4 Performance Bug Fix Size . . . 43
3.5 Threats to Validity . . . 45
3.6 Conclusion . . . 48

4 Performance Mutation Testing 49
4.1 Introduction . . . 49
4.2 Background . . . 51
4.2.1 Performance Mutation Testing . . . 51
4.2.2 PMT Fault Models . . . 52
4.3 SLOWCOACH: A PMT Framework . . . 56
4.3.1 Overview and Workflow . . . 56
4.3.2 Mutation Operators . . . 57
4.3.3 Implementation . . . 62
4.3.4 Prototype Limitations . . . 62
4.4 Evaluation . . . 63
4.4.1 Experimental Setup . . . 63
4.4.2 RQ 1: Mutant Generation and Overheads . . . 65
4.4.3 RQ 2: Functional Equivalence . . . 67
4.4.4 RQ 3: Mutation Score and Discussion . . . 68
4.4.5 Internal and External Validity . . . 72
4.5 Conclusion . . . 73

5 Performance Fuzzing 75
5.1 Introduction . . . 75
5.2 Background . . . 80

5.3 Study Design . . . 83
5.3.1 The Input with Worst Performance and Performance-Size Ratio (PSR) 83
5.3.2 Performance Relevant Input (PRI) . . . 84
5.4 Evaluation . . . 85
5.4.1 Evaluation Setup . . . 85
5.4.2 Overview . . . 85
5.4.3 Discussion . . . 86
5.4.4 Future Work . . . 89
5.5 Conclusion . . . 90

6 Future Work **91**

7 Conclusion **93**

References **97**

Appendix A Case Studies in `grep` **115**
A.1 Case 1 . . . 115
A.2 Case 2 . . . 118
A.3 Case 3 . . . 123

Appendix B Complete Performance Fuzzing Experiment Data **125**
B.1 $\mathbb{S}1$ . . . 126
B.2 $\mathbb{S}2$ . . . 129
B.3 $\mathbb{S}3$ . . . 132
B.4 $\mathbb{S}4$ . . . 135
B.5 $\mathbb{V}1$ . . . 137
B.6 $\mathbb{V}2$ . . . 139

List of figures

1.1 Software Dependency 2
1.2 Threats to Dependability 3
1.3 Kernel Device Driver Example 5
1.4 Performance Testing Procedure 6

2.1 Taxonomy of Performance Issues 14
2.2 Proposed `llvm` Based Profiler 15
2.3 A Flame Graph Example 17

3.1 Distribution of performance bug patterns across all investigated commits . . 40
3.2 Distribution of the identified performance bug patterns relative to the number of investigated commits (stated on top of the bars) for each project. 40
3.3 Performance bug fix duration for different bug patterns measured by FTCF (see Section 3.2.3) 42
3.4 Seniority of bug fixing developers across performance bug patterns. The metric captures the distance of a project-local seniority metric from the median seniority of all candidate developers for the fix on the same project. A seniority of 0 indicates experience matching the median, positive seniority higher experience, and negative seniority lower experience. 44
3.5 Modified source lines of code per bug fixing commit across performance bug patterns 45
3.6 Relative distribution of source line of code change types (addition, deletion, modification) in performance bug fixes by bug pattern 46

4.1 PMT Fault Models 56
4.2 SLOWCOACH Workflow 57
4.3 Cache Memoization optimization pattern ($\mathbb{Q}$4-B in Table 4.3) 58
4.4 Functional Equivalence by Operators and Programs 66
4.5 Mutation Scores of P-mutants by Operator Types and Programs 70

4.6 Mutation Scores of P-mutants by Operator Types and Programs 74

5.1 Execution Time and Path Length 78
5.2 A simplified common procedure of the fuzz testing 82
5.3 Execution Time (Median) of the Slowest Input by Projects and Setups . . . 86
5.4 PSR of the Slowest Input by Projects and Setups 87
5.5 PRI across Projects and Setups 88

List of tables

3.1 Total commit counts for each project 28

4.1 Performance Mutation Testing vs. Mutation Testing [112] 53
4.2 Evaluation Software Projects 63
4.3 Mutation Operators. 64

5.1 Evaluation Targets 83

Chapter 1

Introduction

1.1 Software Dependability and Security

Software systems and services are permeating in recent years, running on a huge variaty of devices, ranging from smart watches, smart phones to supercomputers and Cloud systems. Software services bring unimaginable convenience to our lives. Daily shopping or entertainment can be easily accessed via smart devices. For example, software running on old handheld cellular phones in 80s and 90s was relatively simple by design, on which user interaction is limited to number pads and no third-party software is allowed. In early 2000s, smart phone precursors, e.g. personal digital assisstant (PDA), allow users to interact with them on a touch screen and developers to extend the functionalities of such portable devices with software development kits (SDKs). In recent years, two notable modern mobile operating systems (OS), Android and iOS, enable smart phones to provide most functionalities that are expected on a desktop system, far beyond a portable phone. According to the Cisco annual report in 2022 [30], there will be 3.6 network devices per capita and two-thirds of the entire population over the world will have Internet access. Moreover, the artificial intelligence (AI) technology has drastically changed software sevices in recent years. Learning based AI technologies, such as machine learning, deep learning or reinforcements learning, see wide applications, ranging from recommendation systems [129, 130, 48] to voice recognition [127, 93, 49], computer vision [147, 77].

Despite the improvement in the functionality, software also becomes bulky and complex by magnitudes at the same time. The Linux kernel as an example, had around 10 000 lines of code (LoC) in 1991, and now 31 495 026 lines in 2022[1]. This increased complexity is multiplied by the software running on the kernel, as shown in Figure 1.1. Users interact

[1] Only C source files and header files are counted. Linux v6.1.

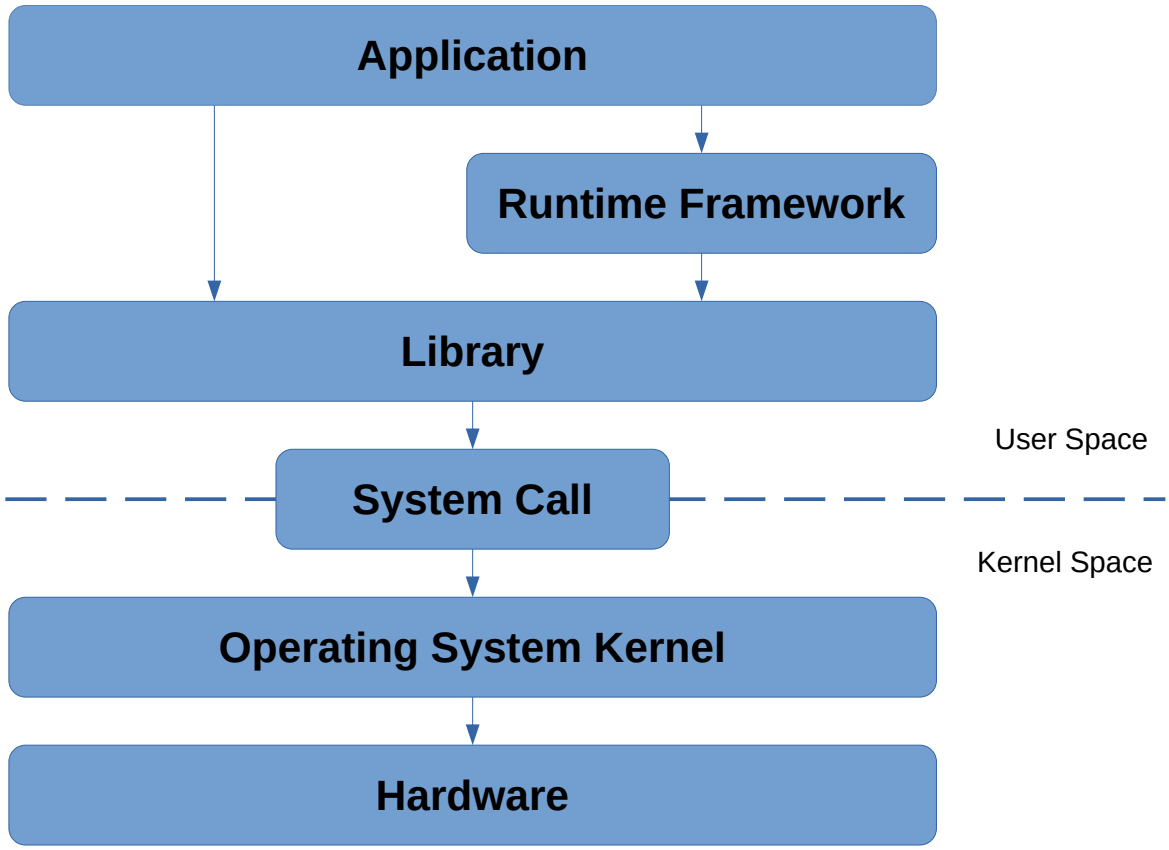


Fig. 1.1 Software Dependency

with applications, which are software programs help users perform activities. Applications need to operate the underlying computer and hence depend on either various libraries to run natively, or some programming language runtimes, e.g., server programs on a Java virtual machine or AI programs by the Python interpreter[2]. Both runtime or libraries eventually request the OS kernel to operate the computer via an interface known as system calls.

If any component in Figure 1.1 is broken, the quality of service (QoS) would be impacted at the user side, as the error would propagate along the dependency chain. Avizienis et al. [10] discussed the problem and software dependability relevant concepts. A correct software system provides *correct services* that behave in accordance to the specification, and any deviation of the specification is considered as a *failure*. Such failure is caused by the deviation of internal states of a program, known as *errors*, and errors are instances activated by *faults*. Figure 1.2 illustrates the causality relations among faults, error and failure. For example, a code snippet contains a bug, and the bug is known as a fault. Given a test suite and some inputs, the buggy code is covered during the test, and the fault is activated. The erroneous

[2]It is possible that applications could directly access system calls, but such cases are very rare.

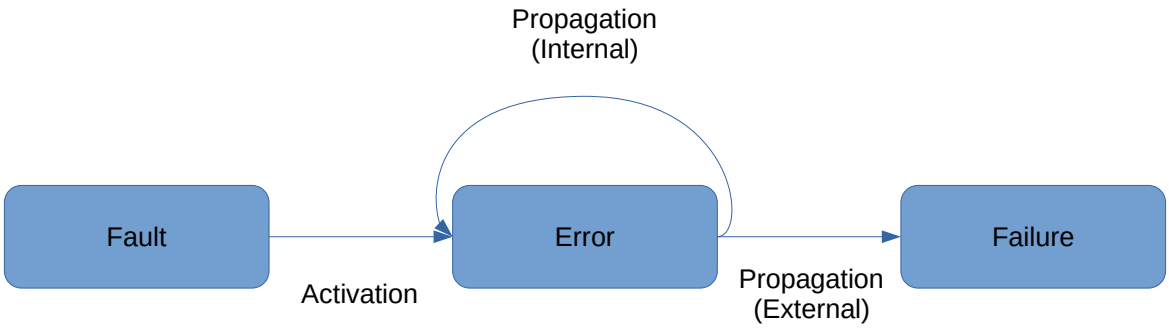


Fig. 1.2 Threats to Dependability

states by the buggy code propagate across the software components until the program crashes. The crash is known as the failure.

Generally, confidentiality, integrity and availability are widely considered as significant security properties, also known as the CIA triad:

- *Confidentiality*: The information of a system is not disclosed to unauthroized third-party.
- *Integrity*: The information of a system is not altered.
- *Availability*: The readiness of a system to provide services with accordance to the specification.

The CIA triad partially overlaps with the attributes to build a dependable system defined by Avizienis et al. [10]:

- *Availability*: The system is available for correct services.
- *Reliability*: The system could continously provide services.
- *Safety*: The failure of the system would not cause catastrophic consequences.
- *Integrity*: The system would not be improperly altered.
- *Maintainability*: The system is able to be repaired and modified.

From the perspective of software, a certain type of software faults could threat different security attributes. In C and C++ for example, a typical fault is invalid memory accesses, such as dereferencing freed memory, which is an undefined behavior (UB) [161, 162, 91]. Usually, such accesses would trap an page fault and the program crashes with a segmentation fault (SIGSEGV) [102, 15, 131], which is a threat to availability. Unfortunately, an out-of-bound

memory access (buffer overflow) could yield more catastrophic results than sheer availability problems, e.g., by allowing attackers to execute arbitraty code [167]. Another (in)famous example is the heartbleed attack, where a single memory copy could unveil credentials and caused massive data breach in 2014 and 2015 [19, 57].

There are many approaches and practices manipulated and adopted to assure the quality of service, such as to build dependable systems and mitigate threats to dependability. In addition to widely accepted software engineering practices such as software design [58, 16, 104] or software testing [101], a lot of state-of-the-art approaches are proposed in the research communities, which will be discussed in detail in Chapter 2. These approaches can be classified into the following 4 categories [10]:

1. *Fault Prevention*: To prevent the introduction of faults. For example, utilizing the right programming language for certain problems, carefully designing the software components, rigorous code review or effective system testing [104].

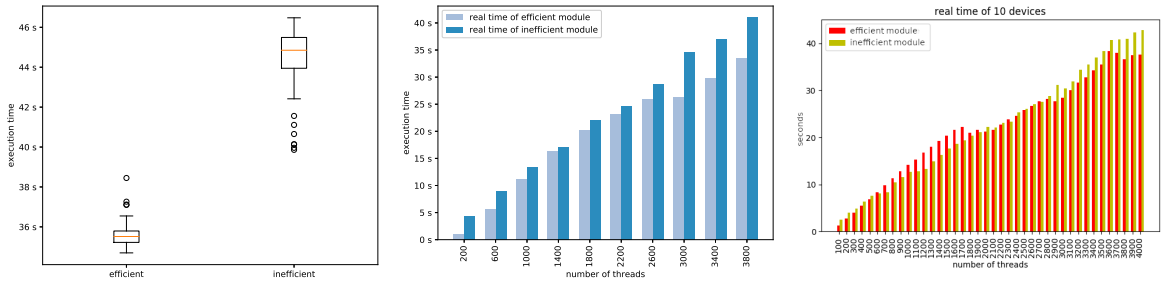
2. *Fault Tolerance*: To keep providing correct services at the presence of faults. For example, microkernels split OS kernels into small chunks in the hope that the system does not crash upon the failure of a single kernel module [149, 150, 74].

3. *Fault Removal*: To reduce the number or severity of faults. For example, a lot of approaches have been proposed to recover from software failures [41, 64, 72, 73, 71]. Recent researches have explored approaches to automatically repair simple faults [79, 81, 80].

4. *Fault Forecasting*: To estimate the number and impacts of faults. For example, Cotroneo et al. [40] predict software aging bugs by the underlying software complexity.

1.2 Performance Testing and Performance Bugs

Many researches mentioned in the previous section [149, 150, 74, 79, 81, 80, 41, 64, 72, 73, 71] are oriented towards functional faults, i.e., the deviation of the system by functional properties. Software systems could also deviate from non-functional properties, e.g., performance bugs, also known as performance issues or performance defects. These bugs can significantly impact the speed and efficiency of a software application. There are a variety of factors causing such bugs, including inefficient code, poorly designed algorithms, and resource contention. The consequences of performance bugs can be severe, ranging from a poor user experience to lost revenue for businesses. In today's digital age, where users expect fast and seamless performance from software applications, it is critical for developers



(a) Execution Time of 4000 Threads on 100 Devices

(b) Execution Time Median on 100 Devices

(c) Execution Time Median on 10 Devices

Fig. 1.3 Kernel Device Driver Example

to prioritize the prevention and resolution of performance bugs. Though Jin et al. [84] define that performance bugs are "software defects where relatively simple source-code changes can significantly speed up software, while preserving functionality", performance bugs can nonetheless be notoriously difficult to identify and fix. There are many reasons causing performance bugs hard to be identified, as performance bugs often do not manifest until certain conditions are met, for example, the application is under heavy load, running for an extended period of time [43, 40, 41] or running under some configuration [69].

To demonstrate performance bugs and performance testing, a series of experiments are carried out, among which one experiment with 100 devices (Figure 1.3a and Figure 1.3b) and the other with 10 devices (Figure 1.3c) are detailed in this chapter[3]. The experiment is composed of two kernel first-in-first-out (fifo) queue implementations resembling the `scull` example of Corbet et al. [37], with the support of blocking accesses from the user space and parameterized number of devices, and a parameterized multi-threaded test program that spawns its threads to read from or write to a given fifo device. The "efficient" module distributes a lock for every device, while the lock is acquired upon reading and writing of the corresponding device until the reading or writing is complete. The "inefficient" module, on the other hand, defines a global lock for all devices which serializes the request processing of all devices. In the user space test program, each thread needs to read or write a fixed amount of data, which means the number of threads represents the workload size. The causality of the inefficient module is similar to the big kernel lock problem in the Linux kernel for years [102, 15, 97].

[3]The original experiments are carried out with 100 to 4000 threads (increment by 100 threads) as workloads running on 10 to 100 devices (increment by 10 devices). Figure 1.3b emphesizes the impacts of performance bugs and it was presented on ECOOP doctoral symposium 2018, which can be found at https://2018.ecoop.org/details/ecoop-issta-2018-doctoral-symposium/11/Testing-for-Performance-Issues-in-OS-Kernels. Figure 1.3b shows part of the experiment results on 100 devices for clarity, while Figure 1.3c is the complete raw results on 10 devices.

Fig. 1.4 Performance Testing Procedure

Figure 1.3 presents the test results of experiments. The y-axis represents the execution time in seconds, or the wall clock time executing the program (collected by the `time` utility), and the x-axis is the number of threads. Each bar in Figure 1.3a represents the execution time when there are x threads reading from and writing to 100 devices. As the figure suggests, the effect of the optimization can make a execution time difference of up to 9 seconds with a maximum relative difference of roughly 300% in the case of 200 threads. However, if the test is repeated 50 times, some repetitions of the "efficient" version only yield 2 s or 3 s improvement, where the best case of the "inefficient" version has only less than one second performance overhead than the worst case of the "efficient" version (see Figure 1.3b).

A typical procedure to fix a performance bug is shown in Figure 1.4. Given the implementation of the inefficient kernel module as the instance of a performance bug case, the first step is to detect the performance bug, which is to determine the existence of performance bugs. In this example, the detection is to compare the performance of the inefficient module with the efficient one as the reference. Then, developers would continue to localize the causality of the performance bottleneck upon the occurence of performance bottlenecks, i.e., to pinpoint where performance bugs dwell in the code, usually by profilers[152, 124]. In the same example, unnecessary lock contention is the causality of the performance bug, and many performance tools, e.g. `perf` [124], `LTTng` [152] or `lockstat` are able to demonstrate where the lock contention happens. Such localization information entails the number of attempts to hold the underlying lock, the time a task waiting for this lock, etc. Most profilers are capable of capturing more information other than locks, and will be discussed in details in Chapter 2. Once the causalities of performance degradation is spotted, developers would analyze the causality and devise the solution to mitigate performance bugs, which is designated as the *code optimization* in Figure 1.4. It is observed that performance bug detection is often strongly coupled with the causality analysis [144, 168, 4], which can be challenging to analyze automatically or with context-free heuristic rules [144, 27, 134].

1.3 Research Questions and Contribution

The efficacy of the performance testing depends largely on the effectiveness of performance detection. A successful detection provides not only the localization of a pathological entity

(a test case and an input or workload), but also hints the code optimization by the causality analysis. In the previous kernel module example, the performance bug detection is based on the reference implementation of an efficient module. In the real world however, there is seldom such a luxury of a reference implementation (a program with optimal performance), but sometimes requirements on the performance in a specification. The performance specifications (requirements) differ on different platforms and scenarios. For example, 1 s latency is usually not a concern for a desktop applications, but could be a hard functional requirement in real-time systems [149]. The performance detection is hence challenging due to the lack of a *test oracle*, which specifies whether the software under test (SUT) is correct or incorrect. Hence, this thesis aims to formalize and address the research problems caused by the lack of testing oracles for performance testing.

Reseach Question 1: ***What are the characteristics of performance bugs?***

In practice, artificial test oracles are usually adopted to identify performance bugs. A prominent example is the regression testing, in which the updated code is tested and compared with the original code during software development cycles [13]. A performance regression is the case when the code change degrades the performance in comparison with that of the original code, where the performance of the original code is the test oracle. Recent researches aim to exploit the opportunities to identify performance bugs more aggresively. Jin et al. [84] for example, proposed a performance bug detection approach that match certain syntactic patterns, such as some functions must be invoked before others, which are specific to a project context. Wen et al. [164], Chabbi et al. [20], Song and Lu [144] proposed approaches based on pathological memory accesses analysis. Unfortunately, the dataset of known performance bugs used by these approaches is no longer available and there is a need for systematic investigation of performance bugs to guide the future research.

Contribution 1: ***A dataset and semantic taxonomy of known performance bugs.***

The missing of known performance bug datasets limits further research on generalized performance bug identification approaches. Hence the first contribution of this thesis is the dataset of performance bug instances in the real-world. Since many performance diagnostic tools are specialized in certain kinds of performance bugs, this dataset is organized by the semantic taxonomy of performance bugs. The classification of known performance bugs help establish performance bug fault models, which will be used for the performance mutation testing (PMT), and detailed in Chapter 4. This dataset can not only assess the

alignment of performance bug models and existing performance diagnostic tools, but also provide a large body of instances for future research evaluation, thanks to its large number of performance bugs studied (over 700 bugs). In addition, this dataset also studied characteristics of performance bugs in terms of performance bug complexity, which are used to prioritize certain types of performance bugs in the future research.

Research Question 2: ***How to determine if the performance testing is well calibrated to identify potential performance problems?***

Another performance bug detection challenge is that, the selection of workload could impact the performance bug detection. In another experiment with identical setups of the previous one, except only 10 devices are available, sometimes the "efficient" module is even less efficient than the "inefficient" module, as demonstrated in Figure 1.3c. The underlying problem is caused by the non-determinism of the lock contention [149], which is also one of the major sources of testing flakiness [98]. The low number of repetitions is another potential problem, as the execution time medium could be statistically biased to check the alignment of the underlying performance with the requirements [110]. An approach to identify correct configurations for performance testing is hence needed.

Contribution 2: ***A performance mutation testing (PMT) framework and a set of useful mutation operators.***

Recent researches aim to verify the correctness of a test suite with *mutation testing* [123, 78]. The mutation testing technique is based on *fault injection*, which aims to inject faults into the SUT to test the robustness of the underlying software. There are two hypotheses on which mutation testing (MT) assumes. One is known as the *competent programmer* hypothesis stating that the buggy code is behaviorally close to the correct code [52], the other is *coupling effects* stating that test suites capable of detecting a certain type of bugs are able to detect more complex bugs [117]. Based on these hypotheses, if injected artificial errors can be detected by a test suite, this test suite should be robust enough to identify more complicated bugs. Mutation testing is therefore to inject faults into the code and to check if test suites are able to identify these faults.

Similarly, performance mutation testing (PMT) injects performance bugs to test if a performance testing suite (benchmark and its workloads) is well calibrated to be able to find mismatch of the SUT performance and performance requirements, if there is any. In Chapter 4, a PMT framework named SLOWCOACH is implemented and evaluated on 4 real-world

projects. As the result of the earlier research (**Contribution 1**) shows, performance bugs are usually difficult to be generalized without contextual information, and therefore general syntactic rules may not be able to generate useful mutants. Chapter 4 hence discusses the taxnomy of performance fault models, based on which SLOWCOACH adopts configurations to embed contextual information into mutation operators.

Research Question 3: ***How to effectively use and adopt performance fuzzing?***

One of the conclusions by the evaluation of the **Contribution 2** is that the workload selection is crucial to the effectiveness of performance testing, as well as performance bug detection. So, automatic approaches to properly generate good performance test cases are expected to be helpful. Thus, the *fuzz testing* technique[4] [169] is worth of discussing in detail.

The procedure of fuzzing is a loop randomly mutating inputs and selecting inputs for the next iteration of random mutation, until the time budget runs out. In classic fuzzing, the inputs to be selected are those either yield larger code coverage, and any inputs crash the SUT are recorded. Lemieux et al. [92] proposed a performance fuzzing framework, known as PERFFUZZ, to automatically generate pathological inputs yielding performance degradation. PERFFUZZ guides the fuzzing by both code coverage and path length (the number of each control flow graph edge, or CFG edge being traversed), where the longer a path length is, the worse its algorithmic cases are explored.

Lemieux et al. [92] have not discussed how PERFFUZZ generated inputs could be used in performance analysis (cf. Figure 1.4), as the evluation focuses on the path length only. Due to the discrepancy between classic fuzzing and performance fuzzing in terms of interesting inputs, the interpretation of interesting inputs would differ greatly. A fuzzing process often generates thousands of inputs, and there is no clear standard to select which to be used for performance analysis, because performance analysis relies on the performance metrics which are not directly provided by fuzzed inputs. In spite of the reasonable approximation of the worst algorithmic case to the "bad" performance of a program, it remains unknown, if a longer path length does yield *interesting inputs*, which are *small in size and large in performance impacts*. Performance impacts can be measured by different dimensions and *execution time* is adopted in this thesis, which is the most direct performance metrics related to the denial of service (DoS) attacks [46]. The measurement of such impacts could be affected by many factors, and the selection of interesting inputs would be affected as a result.

[4]Fuzz testing is also shortened as *fuzzing*. These two terms will be used interchangably throughout the thesis.

Another problem with the path length is the inconsistency of performance costs among CFG edges. Some CFG edges could be slower than others, and hence may not be an ideal fuzzing guidance as admitted by Lemieux et al. [92]. However, it remains unclear, which fuzzing guidances could fit into PERFFUZZ, and what is the efficacy of these fuzzing guidances.

Contribution 3: ***An empirical study on the efficacy of* PERFFUZZ *with various fuzzing configurations and variants based on* PERFFUZZ**

In Chapter 5, an empirical study on the efficacy of PERFFUZZ [92] is carried out. This study comprises of comparisons of 4 performance fuzzing setups, as well as 2 PERFFUZZ based variants. These fuzzing setups encode the aforementioned factors that could affect the results of performance measurements, and eventually the efficacy of performance fuzzing.

The result of comparisons provides advices to explore effective usage of performance fuzzing. The general recommendation to apply performance fuzzing is to customize fuzzing parameters, notably the timeout value as well as the file size limit. The timeout value should be set as large as possible for the fuzzing tool to fully explore generated inputs, and it would nevertheless be dynamically limited by `AFL` based fuzzers, such as PERFFUZZ. The file size limit should be set based on developers' experience, as a too small size limit would confine the fuzzer search space, and a too large one would allow the fuzzer to explore a lot of linearly increasing workloads, which are known to be less interesting. Other conclusions suggest to use custom inputs as well as to keep the compiler optimization levels aligned.

The evaluation on PERFFUZZ argues that the PERFFUZZ framework is limited to generated more interesting inputs with other performance fuzzing guidances. The first limit is that the timeout value is dynamically computed and users can merely set an upper bound. For example, in the experiments on `libjpeg`, the fuzzer usually stops exploring an input if it takes longer than 12 to 13 seconds, despite the manual timeout value is 100 s. The second limit is that performance relevant information is confined within a static program, which is referred as *internal factors*. Moreover, many performance related metrics require contextual information, such as the names of memory allocation functions to guide the fuzzing by memory allocation, as most C/C++ programs implement their own allocators for performance. The result of the second part of the study shows that performance fuzzing timeout and the fuzzing guidance relying on internal factors are the limits to overcome in the future research on performance fuzzing.

1.4 Publications

The following publications have, in parts verbatim, been included in this thesis.

[27] Yiqun Chen, Stefan Winter, and Neeraj Suri. Inferring performance bug patterns from developer commits. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 70–81, 2019. doi: 10.1109/ISSRE.2019.00017

[29] Yiqun Chen, Oliver Schwahn, Roberto Natella, Matthew Bradbury, and Neeraj Suri. Slowcoach: Mutating code to simulate performance bugs. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 274–285, 2022. doi: 10.1109/ISSRE55969.2022.00035

[28] Yiqun Chen, Matthew Bradbury, and Neeraj Suri. Towards effective performance fuzzing. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 128–129, 2022. doi: 10.1109/ISSREW55968.2022.00055

The following publications are related to different aspects covered in this thesis, but have not been included.

[172] Shujie Zhao, Yiqun Chen, Stefan Winter, and Neeraj Suri. Analyzing and improving customer-side cloud security certifiability. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 300–307, 2019. doi: 10.1109/ISSREW.2019.00088

1.5 Organization

This thesis is organized as follows. Chapter 2 discusses the general background related to the performance, existing tools as well as diagnostic approaches. Related work for every forementioned contributions are discussed in further details. Chapter 3 discusses the empirical study on performance bugs across 13 real world C/C++ projects, and the characteristics of performance bugs. Chapter 4 discusses the performance mutation testing (PMT) technique and evaluates it on 4 real world projects. Chapter 5 investigates the performance fuzzing technique. This chapter emphesizes the different performance related scenarios and the limits of applying different fuzzing guidances in current performance fuzzing frameworks. Chapter 6 expands the discussion on the research problems on performance bugs to be addressed in the future. Chapter 7 summarizes the research problems and contributions of this thesis.

Chapter 2

Related Work and Background

This chapter discusses the background information and literature reviews corresponding to Chapter 3, Chapter 4 and Chapter 5. Section 2.1 and Section 2.2 focus on the real-world performance bugs and the technology behind performance bug diagnostic tools. Section 2.1 discusses the general classfication of performance bugs and early research efforts to help address performance problems of this thesis. Section 2.2 discusses existing performance profiling tools and code instrumentation, which is a technique behind many performance diagnostic tools with insights into the runtime information of a program. Section 2.3, Section 2.4 and Section 2.5 review the state-of-the-art performance testing techniques corresponding to Chapter 3, Chapter 4 and Chapter 5 respectively. Section 2.3 discusses the performance bugs and is related to Chapter 3. Section 2.4 discusses trending topics of mutation testing in the software engineering and testing community and is related to Chapter 4. Section 2.5 discusses trending topics of the fuzzing technology in the software engineering and security communities and is related to Chapter 5.

2.1 Performance Issues and Diagnostic Approaches

Performance issues can be generally classified by sympotoms into on and off CPU related, as depicted in Figure 2.1. On-CPU performance issues are usually caused by performance bugs that redundantly computate and fully utilize the CPU [144]. Off-CPU issues on the other hand, indicates unnecessary synchronization events of a program, during which the CPU is not utilized [173]. A prominent cause of off-CPUs events is I/O operations. Given sufficient workload, off-CPUs performance issues could be detected by the CPU utilization.

On-CPU issues can be further divided into CPU-bound or memory-bound performance issues. CPU-bound issues are typically algorithmic problems that redundantly compute unnecessary results [164, 144]. Memory-bound issues are caused by inefficient data accesses,

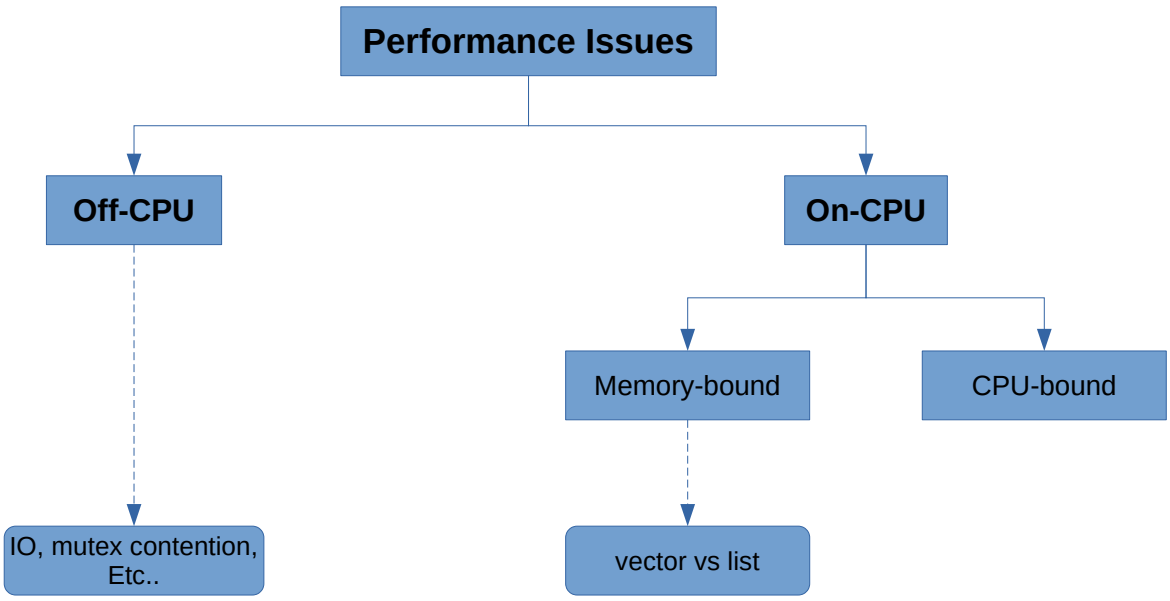


Fig. 2.1 Taxonomy of Performance Issues

due to the huge discrepency of data access performance between CPU and memory [149]. An example of such issues is the time to sum a vector and a list of integers in C++, where the vector based sum needs merely half of the time needed by the list based sum. Memory-bound issues directly impact the effectiveness of performance sensitive services such as game engines or low-latency I/O in FinTech. To mitigate memory-bound issues, developers usually design the software from data perspective in contrast to classic design patterns [138].

The concurrency synchronization problem is between on and off CPU. The lock contention on sleep based locks, such as mutex [97, 102, 15], which put the underlying process (thread) into sleep, are off-CPU events. The contention on polling based locks, such as spin locks, which repeated querying whether the lock is available, are on-CPU events. Mutex contentions can be detected by the CPU utilization, while spin lock contentions can only be detected by tools such as `lockstat`.

In the early stage of this thesis, a profiler was also proposed to capture finer-grained performance details, as detailed in Figure 2.2. The left part of the figure is the on-site tracing. The basic block tracing pass implements the module pass interface provided by LLVM and inserts an instruction to call the profiling function. The profiling function records event traces on `/sys` or `/proc`. The right part of the figure is the construction of the primitive data used for post-processing. After parsing the traced records from `/sys` or `/proc`, a list of events sequence is built with the composition pattern [58]. The traced records are the raw data to be used to detect and localize performance bottlenecks.

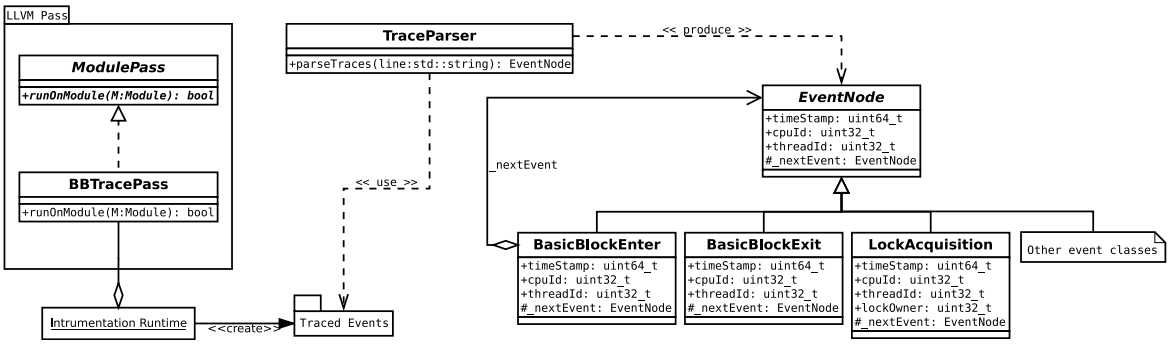


Fig. 2.2 Proposed `llvm` Based Profiler

The proposed approach for detection and localization could then apply various analytic strategies, as the primitive event list is flexible enough for different mutation and aggregation operations. The most intuitive usage of the primitive list is to check the synchronization-working-ratio, which is the proportion of the time spent on interrupt/scheduling/locking events and the time spent on their corresponding parent basic blocks. The synchronization-working-ratio is a useful criterion to detect performance bottlenecks caused by various kinds of overheads, while the suspected basic block is also localized by the profiler. Additionally, the primitive event list provides the information on other processors while a processor is blocked.

Due to overlaps on functionalities seen in other researches [168, 4, 145], this proposed profiler was never published. However, the prototype of this profiler provides valuable information as proof-of-concept for researches in this thesis, e.g., measuring the wall clock time of basic blocks in Chapter 5.

2.2 Code Instrumentation and Performance Profilers

Code profilers are used to demonstrate the performance of a program and its components, e.g., by showing how many seconds a procedure[1] takes or how many times a procedure is invoked. The common technique behind profilers starts by instrumenting small pieces of *probe* code into points of interest (POI) to collect basic performance related information. POIs are usually the entrance and exits of a function or a procedure, and the probe records the information like the time stamp and execution counts. After the execution of a program profilers aggregate the recorded information and visualize the performance overheads of POIs.

[1] In C or C++ it is called a *function*, in Java it is called a *method* and the language agnoistic term used in compilers is a *procedure* [2]. I will use these synonyms interchangeably throughout the thesis.

Since profilers need to intrument probes into code, the profilers are almost always strongly coupled with the underlying programming language and runtime environment. For C and C++ programs, `gprof` [61] is developed based on `gcc` and `X-Ray`[2] [11] is based on `llvm`. Similarly, there is Java VisualVM bundled with JDK and Python has built-in profiling facilities.

Besides programming languages, operating system (OS) kernels also need dedicated profilers due to different runtime environments, as instrumenting code into a running program is technically very complicated[3]. The Linux kernel provides a set of tracepoints that facilitates code instrumentation, performance tracing and kernel events audition [62]. Tracepoints are the POIs defined by kernel developers where the code is relevant to the performance or security properties. For example, the `kprobe` module in the Linux kernel allows developers to dynamically instrument code at the function entrance and exit, as well as lock contentions or other events. Other kernel tracing approaches like `ftrace` [34], `SystemTap` [35] or `LTTng` [152] are also based on the kernel instrumentation and tracepoints. Tracepoints by these tracing subsystems can be dynamically turned on and off to prevent system-wide performance degradation. Besides tracepoints, the Linux kernel also utilizes the hardware features, such as performance monitoring units (PMUs) on X86, as performance counters. One of the most commonly used Linux profiler is called `perf` which samples the CPU time of functions or records these performance counters and visualize the data. `perf` shows not only the percentage of sampled CPU time of each function, but also relative CPU time percentage of functions with the call graph as well as the disassembly code annotated with the CPU time percentage.

In recent years, Linux kernel developers exploit the power of the old Berkley packet filter (BPF) mechanisms to enhance the kernel traceability and observability [63]. BPF was originally designed to allow developers to define the kernel packets handling policies at the user space, e.g., to modify the IP headers when creating a network address translation (NAT) bridge or firewall rules [107, 63]. Linux has an extended BPF subsystem, called eBPF, which allows developers to customize instrumented code with more versatile tracing capabilities. Since BPF is running at user space and defines kernel behavior, any security vulnerabilities of the instrumented code could break the entire OS. Hence, the kernel refuses to load a BPF program with pointers and loops[4]. There are `bcc` that provides easy-to-use programming interfaces along with many predefined performance tracing scripts, and `bpftrace` which is a powerful command line utility with a lot of performance tracing functionalities targeting various kernel scenarios [63].

[2]There is another similar profiler called X-Ray by Attariyan et al. [9]

[3]It is also possible to statically instrument the kernel, but instrumented kernels are too slow to be useful.

[4]Recent patches have added the support for pointers in BPF programs.

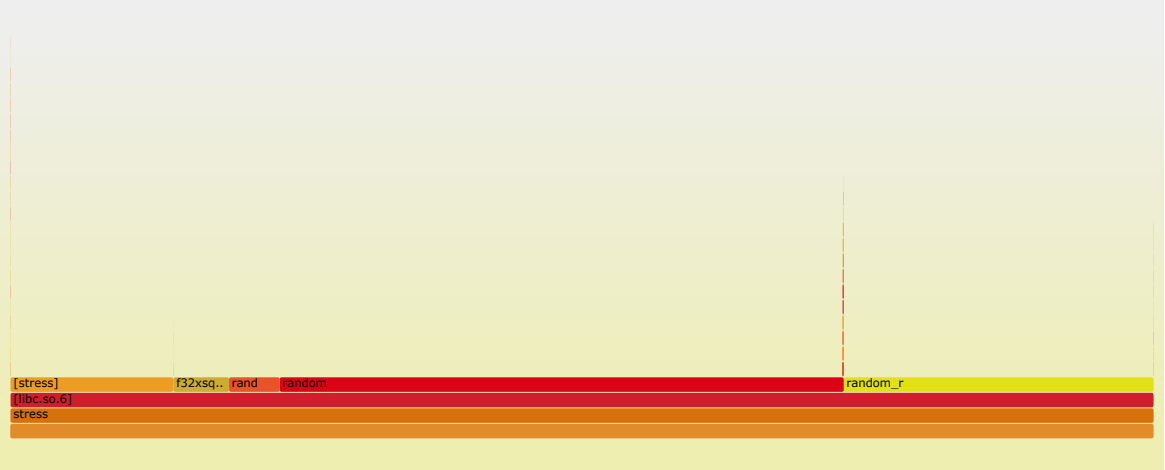


Fig. 2.3 A Flame Graph Example

In addition to the Linux tools, DTrace is a great alternative tracing tool on Unix variants. DTrace was originally developed by SUN for Solaris [103, 105] and adopted by FreeBSD [107] as well as Linux [119] to trace kernel events besides kernel functions. There is even a domain specific language (DSL) known as the D language that allows users to define the instrumented behavior at POIs. Such tracing mechanisms are more generalized and provide finer granular information on the kernel performance, e.g., how many I/Os have been done or how many lock contentions in the kernel have occured.

To help performance bug localization, the data collected by `perf`, BPF based tools or other tools are usually visualized as graphs, e.g., *flame graphs*. Figure 2.3 illustrates such an example, in which the `stress` command tests the CPU with computing sqaure roots of random numbers for one minute[5]. The performance data is the CPU time of each function collected by `perf`. The x-axis in the graph represents the percentage of a performance metric by the function (which is the CPU time in the example), while the y-axis represents the call stack depth. Each bar in the graph is a stack frame (a function), and the wider the bar is, the longer its CPU time is. In the example of Figure 2.3, the performance bottle neck is not the function computing square roots (`f32xsqrt`), but the function computing the random numbers. The CPU time to compute random numbers is more than half of the total CPU time (`random()` + `random_r()`).

[5] `stress -c$(nproc) -t1m`

2.3 Empirical Study on Performance Bugs

2.3.1 Performance Bugs for Evaluating and Training Detection and Localization Approaches

Performance bugs have been studied extensively in recent years and a large number of detection and localization approaches have been proposed. Interestingly, their efficacy has often been evaluated on applications with previously unknown (or disregarded) rather than known performance bugs [87, 85, 118, 4]. A great advantage of such an evaluation is the potential discovery of previously undiscovered bugs. While the detection of every single formerly unknown performance bug is a great achievement, only few bugs are found by each individual approach and bugs from many approaches would need to be combined to form a suitable performance bug data set to evaluate future approaches against. Unfortunately, newly found bugs are commonly described in insufficient detail in (space constrained) research articles to enable their reliable extraction as a reusable performance bug data set, which is the target of the study in Chapter 3.

A number of articles on approaches to detect or localize performance bugs use actual performance bugs or performance bug simulations to demonstrate the aproaches' efficacy [143, 94, 3]. These evaluations commonly focus on few bugs (between 15 and 70 in the cited articles) and do not report the relative occurrence of the targeted bug types compared to other bug types or the complexity of the targeted bugs.

Other articles explicitly state certain performance bug patterns they attempt to detect or localize [142, 26, 156, 146, 155]. Some of these patterns are highly application dependent (e.g., [26]). The patterns in the study in Chapter 3 are mapped to existing patterns based on the information provided in the respective articles.

PerfLearner successfully uses 300 randomly sampled performance bug reports from a total of 1383 reports across three projects to generate performance test frames that are evaluated against 10 other reported performance bugs [68]. The authors of PerfLearner report that extracting and reproducing these 10 performance bugs took approximately 400 work hours, which illustrates the complexity of reproducing performance bugs and relating them with performance bug reports. problems are avoided by directly targeting performance bug fixing commits. A downside of this decision is that there is no way to quantify the performance impact of the bugs in the study. This is believed to be tolerable, as such quantifications are highly sensitive to changes in the studied programs' operational environment, such as hardware and software configurations.

2.3.2 Empirical Studies of Performance Bugs

A number of studies target the assessment and characterization of performance bugs [170, 84, 171, 114, 66] similar to the study in Chapter 3.

Zaman et al. compare quantitative characteristics of performance and security bugs in Firefox [170]. Their automated analysis of more than 180 000 bug reports, out of which 4293 are performance bugs, reveals that performance bugs take more time to fix and are tackled by more experienced developers. The study in Chapter 3 targets a smaller sample of performance bugs, but from a larger variety of projects, which are manually analyzed to confirm they are indeed performance bugs and characterize them according to how they hamper performance. Zaman et al. also present a smaller scale qualitative study of 400 randomly sampled performance and non-performance bug reports from Firefox and Chrome, which reveals that the performance bug reports tend to suffer from poor reproducibility [171].

Jin et al. present an empirical study of 109 randomly sampled performance bug reports from five applications. Their categorization according to how these bugs are fixed resembles the categorization presented in this study with the main difference that the categorization focuses on the intended semantics of performance bug fixes instead of their syntactical appearance, as elaborated in Section 3.2.2.

Nistor et al.'s comparison of 210 performance bugs against non-performance bugs in three software projects shows that performance bug fixes are equally likely to introduce new functional bugs as non-performance bugs and that performance bugs are more difficult to fix than non-performance bugs [114].

Han and Yu conclude from a study of 193 manually inspected performance bug reports and related changelogs across three projects that performance bug observability is highly configuration dependent, while fixing performance bugs does require source changes [66]. This supports the decision to focus the study in Chapter 3 of performance bugs on source code changes.

2.4 Performance Mutation Testing

The term *performance bug* was coined by Jin et al. [84], who investigated more than 100 performance bugs in real-world C/C++ projects and developed a tool for detecting these bugs. Chen et al. [27] surveyed and semantically categorized more than 700 performance bugs from real-world developer commits from 13 popular C/C++ projects. Sánchez et al. [134] investigated the performance bugs across multiple publications in the research community and Tizpaz-Niari et al. [153] surveyed performance bugs in machine learning libraries.

Many approaches based on the symptoms of performance bugs have been proposed. Su et al. [145], Chabbi et al. [20], and Wen et al. [164] detect performance bugs by processor event based sampling (PEBS), which samples hardware events, such as memory or cache accesses, on modern processors. This facility can be used to identify 1. *dead store*, where data are stored to memory but never loaded later, 2. *redundant load*, where data are loaded but never stored back to memory, and 3. *false sharing*, where memory accesses from different threads are close together, resulting in cache thrashing. To detect these bugs with PEBS, performance tests are needed so that dynamic memory accesses can be analyzed. Dynamic memory access patterns are also helpful to detect and localize redundant computation in loops [144]. Moreover, performance bottlenecks caused by off-CPU events [173] and lock contention [148, 4, 168] can be detected using dynamic runtime information. Attariyan et al. [9] propose a more generic state-of-the-art performance profiler to localize performance bugs. State-of-the-practice performance diagnostic tools, e.g., `perf` [124] and `lttng` [152], are capable of profiling both on-cpu and off-cpu events, as well as numerous additional kernel events.

Besides various performance bug diagnostic approaches in different domains, many researchers also consider *computational redundancy* as a significant indicator for performance bugs. Wen et al. [163] define the same return value being computed by repeatedly calling a function as a source of performance bugs. Song and Lu [144] propose a more generic approach to detect whether the results of each iteration are redundant in a loop. Besides logical approaches to detect computational redundancies, Della Toffola et al. [51] aim to find locations where the computational results can be cached by deep learning. The fixing strategies often applied to mitigate redundancy are to either skip the computation, e.g., by introducing a fast path, or cache the results of computation for future usage. Interestingly, these two strategies are among the dominating performance bug fixing patterns in real-world scenarios [27]. Despite many variants of PMT fault models, the computational redundancy anti-patterns from the aforementioned works is adpoted in Chapter 4 since computational redundancies are a stronger indicator for performance bugs in comparison to other metrics.

MT techniques are well studied across the research communities. Papadakis et al. [123] as well as Jia and Harman [78] review the development of MT techniques covering various programming languages from 1970 to 2017. Besides 22 mutation operators for the FORTRAN programming language, Jia and Harman [78] also discussed major problems in the mutation testing. One of the problems is that many mutants are functionally equivalent, which leads to the *equivalent mutant problem* [18]. Such functionally equivalent mutants cause duplicated mutation testing, which gravitates the already high demands on the computational power by the mutation testing procedure itself. In PMT however, the functional equivalence is

considered as a premise for a valid performance mutant, as the performance of a functionally deviated mutant is not a right measure for the performance of a normally executed program. A recent research [99] shows that, from 1979 to 2010, 17 approaches have been proposed, aiming to address the functionally equivalent mutant problem, covering programming languages such as C, Java or FORTRAN. Efforts to eliminate functionally equivalent mutants can be divided into three categories. The first category is the *mutant detection*, which aims to find equivalence by compiler optimization or mathematical constraints. The second one is to avoid functionally equivalent mutants from being generated, e.g., by randomly selecting the mutants to be generated, or by checking logical constraints during the mutation procedure. The third one is to suggest the potential functionality equivalence of mutants. As detailed in Chapter 4, the PMT procedure is complicated (c.f. Figure 4.2). To prevent deviating from the research on PMT itself, the random mutant selection approach is adopted [109] (c.f. Section 4.4.3), meanwhile some hearistics to retain functional equivalence are also applied (c.f. Section 4.3.1). As mentioned by Jia and Harman [78], the functional equivalent mutant problem cannot be completely solved. For the PMT framework in Chapter 4, more functional equivalence check could be applied, as will be discussed in Chapter 6.

Chekam et al. [21] implement and evaluate an early mutation testing framework. Chekam et al. [22] develop a symbolic execution approach to search for the input to kill *hard-to-kill* mutants (please refer to Table 4.1 for definitions). Devroey et al. [53] propose an approach to identify equivalent mutants by NFA simulation. Recent research [125] also confirmed the efficacy of mutation testing in industry practices.

Natella et al. [112] provide an overview of mutation testing techniques in the context of assessing systems against software failures. Research in this area has been focused on the representativeness of injected faults, which is a indicator of whether artificial bugs are hard-to-kill, and thus subtle enough to cause realistic software failures [111]. However, simulating failures through code mutation can be cumbersome, as the mutated program may need to be recompiled, and the mutants have often no effect on the program (i.e., equivalent and hard-to-kill mutants, see Table 4.1). Thus, previous studies have investigated how to efficiently perform mutations on binary code [42], and whether faults injected inside a software component (i.e., through code mutation) can be replaced by more convenient injections at software interface level (i.e., by corrupting data returned from a component) [113]. Other work has also developed a PMT framework with many MT operators [50]. As will be demonstrated in Sections 4.3 and 4.4, SLOWCOACH shows better results in terms of evaluating a performance testing environment by careful implementation of mutation operators and better interpretation of mutation scores. In Chapter 4, efficiency and representativeness issues in the context of performance bug injection are also investigated, by exploring different approaches

from the design space of PMT techniques. Cotroneo and Natella investigate fault injection techniques from a diverse spectrum, e.g., to support the certifiability of software by fault injection [38], the application of fault injection on the Android platform [44], testing binary level mutation efficacy [42], to support the system failure identification in cloud systems [45].

2.5 Performance Fuzzing

The fuzzing technique was first proposed by Miller et al. [108], who found dozens of crashes and hangs in several UNIX systems. In recent years there are enormous work around the fuzzing techniques in the research community, and it is impossible to enumerate all of them. In this section, most representative and relevant work on fuzzing techniques are focused.

AFL [169] is one of the most widely used fuzzer in the research community, as many researches such as the fuzzer discussed in Chapter 5 [92] are based on it. Coppik et al. [36] extended AFL with the guidance that traces memory accesses and found more unique crashes. Böhme et al. [14] proposed AFLGo which targets the modified code coverage for regression testing. AFLGo is guided by a Markov decision process that prioritizes control flow graph (CFG) edges towards the modified code. Chen et al. [24] formalized desired properties for grey-box fuzzing and improved AFLGo. Some programs, e.g. an xml parser, need well formed inputs and many randomly mutated inputs by AFL are rejected. Hence fuzzers are less efficient in these scenarios. Padhye et al. [120] proposed parametric generation of inputs for fuzzing, known as `Zest` to mitigate well-formed inputs problem. Le et al. [90] improved `Zest` by using probalistic heuristics to generate inputs on a grammar. Padhye et al. [121] later generalized the idea of parametric generation by `Zest` and proposed a fuzzing technique based on the domain specific context. Wang et al. [160] proposed a data driven approach to generate seeds for the fuzzing process.

Fuzzing operating system (OS) kernels is another interesting domain. A popular fuzzer for OS kernels is `syzkaller` [60], which synthesizes system calls as inputs for fuzz testing. Schumilo et al. [136] fuzzes OS kernels by hardware virtualization utilities (Processor Trace) [76] and mutate an entire disk image as input. Zou et al. [176] proposed an innovative approach to mutate the network packets as well as system calls and guide the fuzzing by TCP state transitions. This has proven to be effective against vulnerabilities and RCF violations in the TCP/IP stack implementation of the Linux kernel. The fuzzer by Jiang et al. [82] aims to address the problem that error handling code in OS kernels can hardly be covered by test cases as these errors hardly occur at experimental environments. This work injects errors to the kernel so that the error code will always be executed if its precedent code is executed, then fuzzes injected kernels. Jiang et al. [83] later expand this technique to user

space software. Wang et al. [159] applied reinforcements learning to guide the kernel fuzzing. Shi et al. [140] summarizes how industry carries out fuzz testing.

As Crosby and Wallach [46] discussed how algorithmic complexity could impact the performance, many researches explored various static and dynamic approaches in searching for the worst algorithmic cases. Koo et al. [89] use reinforcement learning to continuously update the symbolic execution policies for worst-case test generation. Saumya et al. [135] analyzed branches in a program statically to generate worst-case test inputs. Other approaches [139, 154, 137] investigate synthesized test cases for performance bottlenecks with profiling data. Han et al. [67] investigated the performance bugs reports and developed an SQL based test generation approach.

Besides performance test cases synthesis, there are also many researches on performance fuzzing. Petsios et al. [126] developed a fuzzing technique to identify worst algorithmic cases, based on Clang libFuzzer [33]. Lemieux et al. [92] proposed another performance fuzzing technique that exploits worst algorithmic cases, based on AFL [169]. Liščinský, Matúš [95] developed a performance fuzzer guided by profiling data. More in-depth details on performance fuzzing will be discussed in Section 5.2.

Recently, many researches also investigated the performance impacts of side channel attacks. Allan et al. [7] for example, discussed various of performance related side channel attacks and how can such attacks be utilized for information leakage. Aldaya and Brumley [6] demonstrated the performance degradation caused by side channel attacks can make programs several magnitudes slower.

Chapter 3

Empirical Study on Performance Bugs

Performance bugs, i.e., program source code that is unnecessarily inefficient, have received significant attention by the research community in recent years. A number of empirical studies have investigated how these bugs differ from "ordinary" bugs that cause functional deviations and several approaches [115, 168, 4, 144] to aid their detection, localization, and removal have been proposed. Many of these approaches focus on certain sub-classes of performance bugs, e.g., those resulting from redundant computations or unnecessary synchronization, and the evaluation of their effectiveness is usually limited to a small number of known instances of these bugs. To provide researchers working on performance bug detection and localization techniques with a larger corpus of performance bugs to evaluate against, I conduct a study of more than 700 performance bug fixing commits across 13 popular open source projects written in C and C++ and investigate the relative frequency of bug types as well as their complexity. My results show that many of these fixes follow a small set of bug patterns, that they are contributed by experienced developers, and that the number of lines needed to fix performance bugs is highly project dependent.

3.1 Introduction

Performance is among the most important non-functional properties of programs [88, 166]. Unfortunately, performance bugs accompany the software development process like functional bugs and software quality is equally deteriorated by those, e.g., in the form of decreased end-user interaction, wasted computing resources, or even DoS attacks [92]. A variety of performance diagnosis tools and approaches have emerged over the past decade to assist developers with the identification and localization of a wide range of performance bugs in different scenarios [84, 168, 4, 148]. While it is expected that these tools/approaches cannot eradicate performance bugs entirely, it remains unclear which performance bugs are how well

addressed by these tools or to which degree these tools are being used in practice. Therefore, it is important to investigate which types of performance bugs get fixed (and how) to guide future research in this area.

For this purpose I have conducted a large scale study on 733 performance bug fixing commits across 13 popular open source projects. To illustrate how well existing tools and approaches support developers with the detection and removal of performance bugs, I investigate the duration between the introduction and removal of performance bugs as well as the expertise of the bug fixing developer and the bug complexity in terms of lines modified by the fix. For example, if performance bug detection and removal are well addressed by existing approaches or tools, they must be expected to have a short performance bug fix time. Similarly, with proper tool support, even inexperienced developers should be able to identify and remove performance bugs. Besides these measures, the number of the changed lines in bug fix commits also indicates how complicated performance bug fixes are.

Since tools and approaches for performance bug detection and localization usually specialize for certain classes of performance bugs, I perform a classification of the performance bug fixing commits to assess which classes dominate the bugs being fixed.

Besides assessing the alignment of existing tools and approaches for performance bug detection and localization with characteristics of performance bug fixing commits, I hope that my study can serve as a benchmark that future tools and approaches can be evaluated against, similar to CoREBench [13] or BugSwarm [55] for correctness bugs and I make the entire data set publicly available for this purpose[1]. Despite the relatively large number of subjects in my data set, there are applications that require even larger numbers of defects to obtain statistically significant evaluation results. For these scenarios I envision the creation of *performance mutants*, along the lines proposed in recent work[31] and proposals [133], and provide insights to support the creation of performance mutants that resemble performance bugs fixed in real world projects.

Summing up the above discussion, my study makes the following contributions:

- I present results from a large scale study of 13 open source projects from different domains with a total of 733 manually analyzed performance bug fixing commits.
- Using data from this study I assess the alignment of the current state of the art in performance bug detection and localization with performance bugs that get fixed in practice.

[1]https://yqchen.gitlab.io/perf-bugs/

- The result of my study provides a database of performance bug fixes to serve as a benchmark for the further development and improvement of performance bug detection and localization techniques.
- The discussion on how the presented work can serve as the basis for performance mutation operators, but also why this basis is not (yet) sufficient for a practical performance mutation approach.

The remainder of the chapter is structured as follows. After reviewing related work in Section 2.3, I discuss the methodology adopted in this study in Section 3.2. Then I will discuss the categorization of performance bug code patterns in Section 3.3, followed by an analysis of other performance bug fixes (fix duration, developer experience, changed lines) in Section 3.4, a discussion of the threats to validity in Section 3.5, and a conclusion in Section 3.6.

3.2 Methodology

I investigate performance bugs in real-world projects to assess how well performance bugs targeted by detection and localization approaches are aligned with the bugs that get fixed in practice. Due to the difficulties that performance bug reproducibility poses (see Section 2.3), I identify performance bug fixing commits by manual inspection. To limit the corresponding overhead I pre-filter and sample commits according to criteria discussed in Section 3.2.1. Section 3.2.2 details how the identified performance bugs are classified and how this entails deviations of the derived taxonomy from existing ones. I then introduce the performance bug complexity metrics I use in this study: a measure of performance bug fix duration in Section 3.2.3, a measure of experience for performance bug fixing developers in Section 3.2.4, and a bug fix complexity measure in Section 3.2.5.

3.2.1 Selection of Projects and Commits

I start the choice of target projects in my study from the top 100 popular projects from the Debian repository[3] that are written in the C programming language. I base my selection on the "vote" data of Debian's popularity contest, which reflects the regular usage of the projects. My focus on C is motivated by the observation that performance critical code is commonly written in languages that are "close" to the underlying hardware platform and that

[2]The project name for the next Firefox version in development
[3]https://popcon.debian.org/by_vote

Project	Total
NetworkManager	209
pulseaudio	106
grep	123
rsyslog	136
lvm2	123
llvm	4567
git	1107
clang	860
gecko-dev[2]	4329
openssl	169
systemd	327
libgcrypt	145
linux	18975

Table 3.1 Total commit counts for each project

compile to native machine code. Moreover, as a language that dominates operating systems and other important parts of virtually every software stack, C has a high practical relevance. This is also reflected by the observation that the top 100 C projects in Debian's popularity contest are among the 133 top projects when no restriction is made on the programming language. In addition to these projects, `clang`, `llvm`, and `linux` are also selected as survey targets, because they are widely used large and complex projects with numerous commits and contributors. Thus, the performance of these three projects is expected to be of relevance for a large user base.

I identify performance bug fixing commits in the selected projects by searching for a number of keywords in the commit messages, as listed below along with matching text examples.

- **performance** "This patch improves the **performance** by ... %"
- **speed up** "These changes **speed up** the processing of"
- **accelerate** "This patch **accelerate**s"
- **fast** "After the patch it is ... times **fast**er"

- **slow** “Before the patch it is **slow** in function”
- **latenc** “The **latenc**y of . . . is reduced ”
- **contention** “This patch reduces the **contention** of”
- **optimiz** “The **optimiz**ation of the function”
- **efficient** “The patch makes function . . . more **efficient**”

I also exclude those projects from my candidate list, (1). for which I cannot easily access commit messages because I cannot unambiguously identify or access the official development repository (21 projects), (2). that are Debian specific and not used on other distributions in order to avoid a corresponding bias (4 projects), and (3). that have less than 100 commits that match my keywords (65 projects).

The last of these criteria has been added to exclude projects that do not have a particular performance relevance. For instance, `libcap2` implements operations to get and set POSIX capability states, which are not particularly performance critical. Accordingly, the project does not have a single commit message matching the aforementioned keywords and is, hence, excluded from my study. The 13 projects meeting all criteria are listed in Table 3.1 along with the total matching commit counts. The complete table with matching counts for each keyword can be found on the my data set website provided in Section 3.1.

The keyword-based detection mechanism for performance bug fixing commits is susceptive to false positives, e.g., **performance** could also match a feature commit stating “This patch does not introduce a **performance** regression”. Hence, all matched commits need manual investigation. As some the projects like `clang`, `linux`, or `gecko-dev` have thousands of matching commits, I limit my manual assessment to a random sample of 200 commits for those projects.

3.2.2 Taxonomy

Performance bugs can be categorized by various criteria, e.g. the work by Jin et al. [84] studies syntactical representations of performance bugs. This study, instead, classifies performance bug fixes by the semantics behind these code changes. For instance, the code in Listing 2 shows the introduction of a new API from the syntactical perspective. The goal of the taxonomy developed in this chapter, in contrast, focuses on *how the newly introduced function and its usage affect the performance of the implementation*. In Listing 2, the newly introduced `try_fgrep_pattern()` function introduces a more efficient matcher, which only applies for certain scenarios. If such a scenario is encountered, this light weight matcher

provides a faster execution path that speeds up the character pattern matching process. Therefore, this commit is tagged "fast-path" as the code change introduces a shortcut to speed up execution for certain scenarios (a detailed discussion is provided in Section 3.3.1).

I give preference to a manual semantic classification of performance bug fixing commits over a purely syntactical taxonomy to obtain comparability of bug fixes that transcend project or developer specific preferences, such as coding styles, to which purely syntactical taxonomies are sensitive.

3.2.3 Bug Fix Time

To determine the need for better tool support, I extract a number of metrics in my analysis of the identified commits. The first metric captures the latency to fix performance bugs. Intuitively, the more time developers spend on fixing a performance bug, the more difficult the performance bug is. However, this metric may be misleading in a cross-project comparison, as projects evolve at different speeds. For example, if project A has hundreds of commits every day while project B has only a handful of commits every week, the performance bugs in project B are likely to take longer than in project A, although the performance bugs in project B may not be any easier to detect and fix than those in project A. Consequently, I also take the number of commits between the introduction and the fixing of performance bugs in a project into consideration, to determine whether the project is actively maintained. I define the metric *fix time commit frequency* (FTCF) to present the accordingly normalized fix time as:

$$FTCF = n_{cmt}(\,(t_{intro}, t_{fix}]\,) \tag{3.1}$$

where t_{fix} is the time stamp of the fix commit, t_{intro} is the time stamp of the commit which introduces the fixed performance bug, and n_{cmt} denotes the number of commits in the specified interval. The larger the FTCF value is, the longer is the fix time if projects have the same level of activity in terms of commit rates. If a project has a commit rate that is twice as high as that of another project, then its FTCF is also twice as high for an identical time interval, indicating that the corresponding bug fixing time is actually slower. The reasoning behind this is that in intensively maintained projects, bugs should also be detected and fixed faster. To obtain the FTCF, I need to identify the bug introducing commit that corresponds to an identified fix. I basically follow the widely used approach to infer bug inducing commits outlined in [141], but assume that each modification in the bug fixing commit is *a necessary modification* to fix the bug. Consequently, I search for each modified line in the bug fixing commit the commit that last changed that line before. To be conservative and rather under- than over-estimate the actual fix time interval, I select the commit that is temporally closest

to the fixing commit from this set as the bug introducing commit, because that is the last commit that made a change that then required a fix.

3.2.4 Seniority of Fixers

I also consider the expertise of the developers, who are fixing performance bugs, as an indicator of the fixing effort. If performance bugs are mostly fixed by more experienced developers, this indicates that better tool support for the detection and localization of performance bugs may be required or that the existing tools require a high expertise to be used effectively. To quantify the developer expertise, I measure the time Δ_{dev} between the bug fix at time t_{fix} and the first commit of the fixing developer dev at time $t_{dev(1)}$.

$$\Delta_{dev} = t_{fix} - t_{dev(1)} \tag{3.2}$$

By comparing the experience Δ_{dev} among the developers in the project, it is clear whether the fixer of performance bugs are relatively more experienced in the project or not. The comparison should only cover those developers who are actively contributing to the project at the time of the fix. Thus, the set of developers, who are *candidates* for contributing the fix, is:

$$C_{dev} = \{dev | t_{dev(1)} \leq t_{fix} \leq t_{dev(n)}\} \tag{3.3}$$

where $t_{dev(n)}$ is the time stamp of the last commit contributed to the project by developer dev. Using the expertise measured as Δ_{dev} across all developer candidates $dev \in C_{dev}$, I can judge the relative expertise of a developer contributing a bug fix within a project. However, using such project wise ranks may not be accurate in reflecting the skill of bug fixers in a cross project comparison, because the difference in the total number of developers varies significantly among the projects. For instance, assuming fixer A is ranked 10 out of 20 developers in a project, while fixer B is ranked 500 out of 1000 developers in another project, fixer A may not be more skilled than fixer B despite the higher absolute rank number. Moreover, the absolute value of the time difference is obviously misleading as projects are started at different times and, thus, have different lifetimes, which can introduce significant offsets in the Δ_{dev} values that are more strongly affected by the project than the actual developer experience. To quantify the skill of bug fixing developers and make them comparable across projects, I introduce a project *seniority* metric to evaluate the skill. The first concern of the seniority is the time when the performance bug is fixed and when is the project started. The time difference between the project initialization and the bug fix is noted

as Δ_{base} and defined as:

$$\Delta_{base} = t_{fix} - t_{init} \tag{3.4}$$

where t_{init} is the time stamp of the first commit in the project. The seniority of dev at t_{fix} is defined as:

$$S_{dev} = \frac{\Delta_{dev}}{\Delta_{base}} = \frac{t_{fix} - t_{dev(1)}}{t_{fix} - t_{init}} \tag{3.5}$$

Given a bug fix commit, the skill of the fixer can be represented as the seniority. To have a relative comparison across the project, the seniority of the fixer denoted as S_{fix} is compared to a mean value of the seniority of all developers, given a performance bug fixing commit. In my study I select the median to aggregate the seniority vector of a project and the seniority difference is defined as:

$$\Delta S = S_{fix} - median(\{S_{dev} | dev \in C_{dev}\}) \tag{3.6}$$

This seniority measure reflects the experience of the bug fixing developer relative to the first commit of the project and relative to other active developers in the project and is suitable for a cross-project comparison.

3.2.5 Number of Changed Lines

The number of changed lines directly indicates how complicated a bug fixing commit is. Usually simple changes with a small number of lines modified are more likely to be diagnosed by tools. If a pattern of performance bugs involves a lot of small sized commits, the current tool support probably has not covered this form of performance bugs yet and new tools on such problems are needed. The number of changed lines is also relevant for my longer term goal to create performance mutants to test performance bug detection and localization approaches against, because they indicate to which degree traditional mutation operators (and their implementation in mutation tools) that are commonly applied to individual statements of a program are sufficient to simulate realistic performance bugs. Similarly, the type of change is giving an indication about the nature of performance mutation operators. If performance bug fixes tend to add rather than remove lines, the corresponding mutation operators would need to remove parts of the source code in order to introduce performance bugs. Otherwise, mutation operators would have to insert buggy code to create performance mutants. Therefore, I provide detailed data for the studied bug fixing commits, i.e., the total number of affected lines along with ratios of added, removed, and modified lines.

3.2.6 Bug Collections

I expect my data set of fixed performance bugs to serve as a basis for the evaluation of future performance bug detection and localization approaches, similar to existing data sets for correctness bugs. CoREBench [13] is a collection of 70 regression errors from four open source projects that have been extracted with the goal to serve as a more realistic alternative to mainly hand-seeded bugs in the Siemens test suite [56] and the SIR [75]. BugSwarm [55] is a collection of Python and Java correctness bug fixes mined from Travis-CI logs of GitHub projects and reproduced in isolated environments. Defects4j [86] is a dataset of 357 real world Java functional bugs. Contrary to my data set, none of the three projects ensure that each of their bugs are reproducible. The reason why I cannot guarantee reproducibility is that the magnitude of latencies induced by performance bugs heavily depends on (a). configuration parameters of the software project [66] and (b). the complexity of inputs used to trigger the bugs [92]. It is likely for the same reasons that performance bugs are reported to be more commonly detected and fixed via code reasoning than dynamic tests and that they sometimes "magically" disappear [114]. I have, therefore, decided to focus on bug fixing commits rather than bug reports in my study. None of CoREBench, BugSwarm and Defects4j cover performance bugs.

3.3 The Shape and Variety of Fixed Performance Bugs

During the manual investigation of the commits identified in my keyword-based search, I found that many performance bug fixing commits follow certain patterns. Based on this observation, 7 common patterns have been identified. Among these patterns, two (*asm* and *async*) are project specific and require detailed knowledge of the respective subsystems in the project. These patterns are not likely to be found in other projects and introduce a certain project-based bias to the presented results. Therefore, the discussion of these two categories is kept brief and combined with the discussion of the generic *misc* class of bug fixing commits that do not match any of the larger pattern classes in Section 3.3.6.

3.3.1 Fast-path

A *fast-path* is a construct to avoid repeated or slow computation when possible. I limit the fast-path notion in this study to control flow based fast-paths and classify other avoidance techniques as other patterns. The control flow based fast-path pattern can have different syntactic representations. A simple form of skipping heavy computation is demonstrated in Listing 5. The `if` statement is a typical fast-path avoiding heavy computations when they are

not needed. Real-world occurrences of this pattern are usually much more complicated and

```
int foo(int bar) {
  if (some_cond(bar))
    return fast_path();
  return very_heavy_computation(bar);
}
```

Listing 1 Simple fast-path example

obfuscated. For instance, fast paths may be needed in loops, where programs tend to spend most of their time [2, p. 655]. If heavy computations are encapsulated in functions that are called within a loop body, existing profilers cannot identify the inefficient code inside loops, as profilers rank functions with aggregated execution time. Hence, tools to analyze loops are helpful to identify such cases. Nistor et al. studied memory access patterns and proposed Toddler to detect inefficient loops [115] while Song et al. detect inefficient loops more effectively by combining both static and dynamic analysis on root causes [144]. Tsakiltsidis et al.[157] listed a string of python anti-patterns, which include a couple of examples that I classify as fast-path, e.g. using `if` branches to circumvent heavy computations imposed by logging. To avoid slow-path execution, developers usually have to manually implement the fast-path and ensure that both paths are functionally identical. A fast-path implementation may, for instance, apply a different algorithm to achieve the same result as the slow-path. An example is commit `290ca116c917`[4] in the `grep` project. Listing 2 shows a simplified diff of this commit. The commit fixes a performance regression when multiple regular expression patterns are provided to the program. The generic matcher instance matches slowly, which is why the function `try_fgrep_pattern()` is implemented to "peek" if provided regular expressions can be matched by a light-weight matcher. If they can, the code simply uses the light-weight matcher and else falls back to the generic one.

3.3.2 Arguments

Some commits change the values of arguments passed to a function so that the control flow can take an existing fast-path in that function. In a more generic sense, this pattern represents those optimization that bypass heavy or redundant computations by controlling the input value. For instance, when the input value for `bar` in Listing 5 is chosen so that

[4] http://git.savannah.gnu.org/cgit/grep.git/commit/?id=290ca116c9172d97b2b026951fac722d3bd3ced9

```
+ static int
+ try_fgrep_pattern(int matcher,
+                   char *keys,
+                   size_t *len_p) {
+   /* Implementation */
+ }
  int main(int argc, char **argv) {
    ...
+   else if ((matcher == G_MATCHER_INDEX ||
+             matcher == E_MATCHER_INDEX)
+            && 1 < n_patterns)
+     matcher =
+       try_fgrep_pattern(matcher,
+                         keys, &keycc);
    execute = matchers[matcher].execute;
    ...
  }
```

Listing 2 Example in `grep`

`some_cond(bar)` is more likely to return a non-zero value, the performance of function `foo()` would improve.

As we will see in Section 3.4, this pattern occurs in all targeted projects. I observed that many operations are controlled by flag arguments, i.e., bit fields passed to the processing function to control its behavior. The flag passed to the `do_fork()` function in the Linux kernel, for example, specifies whether the processing fork should copy the page table or file descriptors etc. By setting or clearing bits in the flag, the function execution may behave differently, including the execution of a fast-path instead of a slow-path. In a broader sense, any global state of the program can also be regarded as arguments passed to every function in the program. Thus, like fast-path pattern in Section 3.3.1, arguments related performance bugs also have a wide range of syntactical representations and often require complex reasoning to set arguments or global variables in a way that improves performance, but does not entail functional deviations.

3.3.3 Cache memoization

The result of a computation should be stored if it is needed onward to avoid redundant re-computations of the same result. A trivial loop iterating over a string like in the following example is unnecessarily slow when the code is compiled without compiler optimizations.

```
for (char *c = str;
     c - str < strlen(str);
     c++) { /*...*/ }
```

The loop keeps calculating the (unchanged) length of the dynamically allocated `str` to test whether the iterator has reached the end of the string. Although most modern compilers can move the call to `strlen()` outside the loop body by performing a loop invariant analysis [2, p. 641], the optimization does not cover all cases. If the duplicated calls to `strlen()` are hidden in a wrapper function, the success of the compiler optimization depends on the wrapper returning an invariant value and the ability of the optimization to infer that. I refer to performance bug fixes that "cache" the result of such computations for future usages as *cache memoization*. Intuitively, cache memoization effectively solves the redundancy in the previous duplicated string length computation example. The pattern name is coined in the work by Toffola et al. [51], which lists opportunities to cache computation results in JavaScript.

In C/C++ projects, cache memoization optimizations are also frequently observed. The commit `3548068c22f8`[5] in `clang` exemplifies a typical cache memoization situation. All modifications are applied to the `Sema` class (semantics) of the Objective C frontend. The patch adds a selector variable named `RespondsToSelectorSel` in `Sema` to cache a selector (not shown in Listing 3) and modifies a callback function (`Sema::ActOnInstanceMessage()`) in Listing 3. The callback tests if the event related selector `Sel` equals the contextual unary selector and removes the selector from the warning pool in the case of equality. Instead of fetching the contextual unary selector every time the relevant event is fired, the optimized version tests the equality of `Sel` and the cached contextual selector, and fetches the contextual selector only when it is not yet cached.

3.3.4 Data Access

Different data structures yield different data access overheads, e.g., retrieving unsorted data without an index in a vector or list requires a linear search, while accessing data from hashed maps only introduces the overhead of hash functions. To speed up data accesses, many projects provide developers with a set of predefined data structures optimized for project specific usages. However, depending on the complexity of the project and the variety of data structures, it may not be easy for developers to anticipate how these data structures are best used during development. In `llvm` for instance, `llvm::DenseMap` pre-allocates a large bulk

[5] git commit id: 3548068c22f809e5bc64b83d2c3622018469256c

```
- IdentifierInfo *SelectorId =
-   &Context.Idents.get("resp");
- if (Sel ==
-   Context.Selectors.
-   getUnarySelector(SelectorId))
+ if (RespondsToSelectorSel.isNull()) {
+   IdentifierInfo *SelectorId =
+     &Context.Idents.get("resp");
+   RespondsToSelectorSel =
+     Context.Selectors.
+     getUnarySelector(SelectorId);
+ }
+ if (Sel == RespondsToSelectorSel))})
    // remove selector
```

Listing 3 A simplified cache memoization example in `clang`

of memory for faster iteration on small key value pairs[6]. As a substitute of `std::map<KeyT, ValT>` from the standard library, `DenseMap` yields better performance for the case in commit `f28cb39e4ca0`[7], as shown in Listing 4. The effect of such a change is usually difficult to predict upfront without intimate knowledge of the data structure and the context in which it is used. I assume that it is for the same reasons that this category of performance bug fixes is less covered in existing work.

```
  class GlobalsModRef : public ModulePass,
    public AliasAnalysis {
    // ...
-   std::map<const Value *,
-            const GlobalValue *>
-     AllocsForIndirectGlobals;
+   DenseMap<const Value *,
+            const GlobalValue *>
+     AllocsForIndirectGlobals;
```

Listing 4 A simplified `DenseMap` example in `llvm`

[6] http://llvm.org/docs/ProgrammersManual.html#llvm-adt-densemap-h
[7] git commit: f28cb39e4ca07c387dd270ce123753f898a75d5c

3.3.5 Synchronization

Multicore processors have brought significant performance boosts for parallel and parallelizable programs. Despite the processing speedup, multiple processors accessing the shared memory simultaneously have raised the problem of possible race conditions. To surmount the problem, memory accesses are synchronized by *synchronization primitives* to guard *critical sections*, in which accesses to shared memory are serialized. As serialization diminishes the performance gains from parallel processing, improperly serialized program parts can become performance bottlenecks, e.g., when a critical section protects thread private data (e.g., stack objects) or the critical section is protected by inefficient synchronization primitives, as these primitives themselves yield differing overheads.

In C/C++ projects, mutex-like locks are the most commonly used synchronization primitives, while *spinlocks* are widely used in operating system kernels. Since a mutex may block the execution of the program, low CPU utilization could be regarded as a rudimentary indicator of possible performance problems [168]. Besides CPU utilization, developers nowadays also profile the waiting time of each thread on a lock [4, 148] to localize where locks are mostly contended.

In the projects I assess in this study (as we will see in Section 3.4.1), synchronization related performance problems are relatively infrequent compared to other performance bug fixes (but also take more effort to fix). My investigation in related projects shows that developers nowadays tend to minimize the amount of shared data to avoid race conditions. Linux kernel developers also tend to use the lockless RCU (Read-Copy-Update) mechanism [106] to prevent heavy weight synchronization.

3.3.6 Miscellaneous

Some of the targeted projects introduce low level assembly implementation of algorithms. I refer to such optimization as the *asm* pattern. In particular, cryptographic libraries rely on the fine tuned inline assembly implementations to utilize CPU specific hardware features for speeding up en- and decryption operations. In the two cryptographic libraries I investigate in this chapter (textttlibgcrypt and `openssl`) there are profound optimization patches using new CPU features in these two projects[8] as demonstrated in Figure 3.2. Apart from cryptographic libraries, the Linux kernel also features optimizations of inlined assembly in the sampled commits.

[8] `openssl` uses perl for inlined assembly, so 139 commits involving inlined assembly are not counted in my statistics

Another less common pattern is the optimization for I/O heavy scenarios. To avoid the time spent on blocking I/O, the optimization uses non-blocking counterparts of the I/O operations and waits for the operation in an asynchronous handler. This pattern is thus labeled *async* and is observed in `systemd` and `NetworkManager`.

Some commits apply fundamental changes to a project to improve performance. Such commits involve changes of the software architecture and highly rely on specific contextual knowledge of the project. Therefore, this category is of limited relevance for the goal of my study and I do not discuss this category in further detail.

3.4 Performance Bugs Characteristics

In the following I discuss the results of my empirical study of 733 manually investigated performance bug fixing commits from 13 open source projects. The distribution of performance bug patterns is discussed in Section 3.4.1, followed by a discussion of the effort for fixing performance bugs in Section 3.4.2, Section 3.4.3 and Section 3.4.4.

3.4.1 Bug Pattern Distribution

To assess the relative frequency of different performance bug fixes, I categorize all fixes according to the patterns described in Section 3.3. Figure 3.1 shows the result of the classification across all investigated performance bug fixing commits and Figure 3.2 shows the pattern distribution for each project. The number on each bar is the number of performance bug fixing commits for the respective project.

From Figure 3.1, I observe that the most dominant form of fixed performance bugs is the "fast-path", which accounts for 43 % of all sampled commits in my survey. As discussed in Section 3.3.1, this pattern corresponds to a wide range of syntactical representations and, intuitively, fast-path is a straightforward way to circumvent slow operations. The second most frequent category is composed of the idiosyncratic performance bugs that do not match any common pattern. The *argument* pattern is the third most frequent pattern contributing 14 % of all performance bug fixes. As discussed in Section 3.3.2, C and C++ developers often use flags to control dynamic behavior and, thus, tweaking flag arguments passed functions can also optimize the performance.

Surprisingly, performance bug fixes involving inline assembly language account for 10 % of all investigated bug fixing commits. However, as Figure 3.2 shows, the assembly pattern fixes only occur in three projects, with a strong majority in a single project, i.e., `libgcrypt`. In `libgcrypt` the most frequent performance optimizations are gained by utilizing new

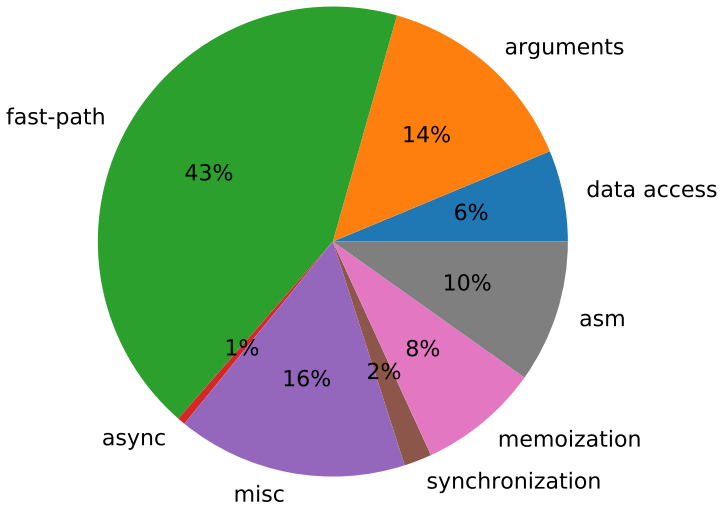


Fig. 3.1 Distribution of performance bug patterns across all investigated commits

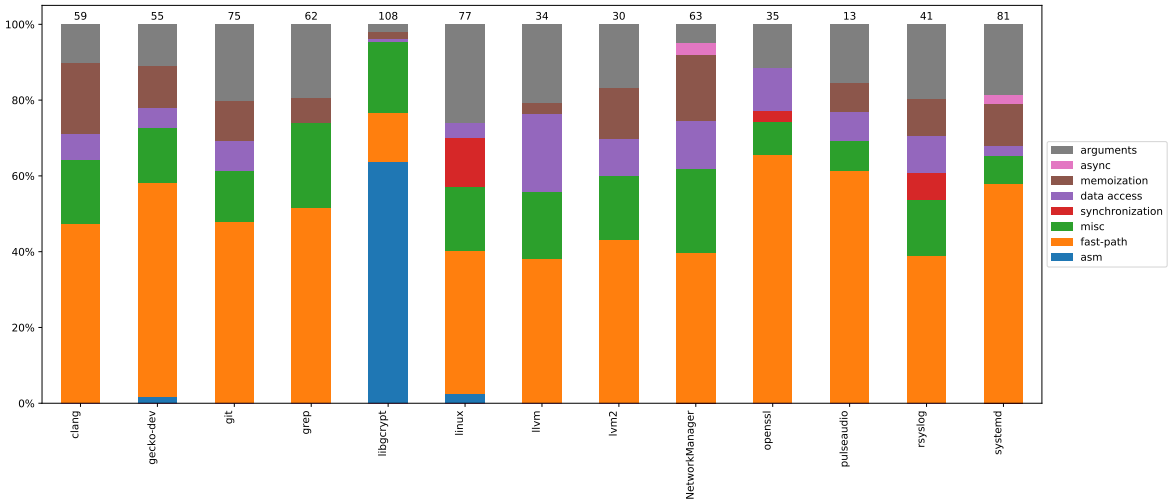


Fig. 3.2 Distribution of the identified performance bug patterns relative to the number of investigated commits (stated on top of the bars) for each project.

CPU features to boost various cryptographic algorithms. The few assembly optimizations in `linux` involve subtle fixes of the crucial procedures written in assembly. From these observations I conclude that assembly based performance bug fixes are strictly limited to very specific application scenarios.

The cache memoization and data access take the most of the remaining code pattern shares, accounting for 8 % and 6 % of the patterns. Although these numbers are not particularly high, it is important to note that the patterns occur across almost all projects in my investigation.

Synchronization problems are the least common performance bug pattern fixed in the investigated commits. Synchronization related performance bugs, in particular those related to lock contention, have been addressed by previous research [168, 4, 148]. This work, however, is relatively new and I do not expect the developed techniques to be already part of the standard tool set of open source developers. From the commits I investigated, my impression is that synchronization related performance bugs either are a relatively rare occurrence or that they just do not get detected and fixed. Firefox, for instance, was once a project suffering synchronization related performance bugs [168] while in my investigation no synchronization problems have been sampled. Another example is `linux`, where the synchronization fixes in the sampled commits are not related to lock contentions, but substitute locks with lockless RCUs [106].

The least frequent performance bug fix pattern I observed is to make tasks asynchronous. Such optimizations only apply for very specific scenarios, where either some procedures are I/O heavy or the result of the procedure is not needed for some time.

3.4.2 Performance Bug Fix Duration

The effort devoted to fixing performance bugs is a significant indicator to prioritize performance problems to be addressed in future research. As discussed in Section 3.2.3, I use the number of commits between the bug introducing and fixing commits to indicate the performance bug fix duration. Figure 3.3 shows this number (FTCF) across various patterns, where the y-axis is scaled logarithmically due to some outliers with high values.

The boxes in Figure 3.3 show that the median fix time of most patterns lies between 10 and 100 commits, with the exception of asm and cache memoization. As discussed in Section 3.3.6, asm basically utilizes new CPU features. Based on my observation that most commits in the asm category replace less efficient assembly instructions by more efficient ones, this indicates that applying new CPU instructions to improve assembly code performance needs less time than optimizing inefficient code written in a higher order language. Another quickly fixed bug pattern is cache memoization. Therefore, redundant computations of invariant results seem to be easy to identify and straight-forward to repair.

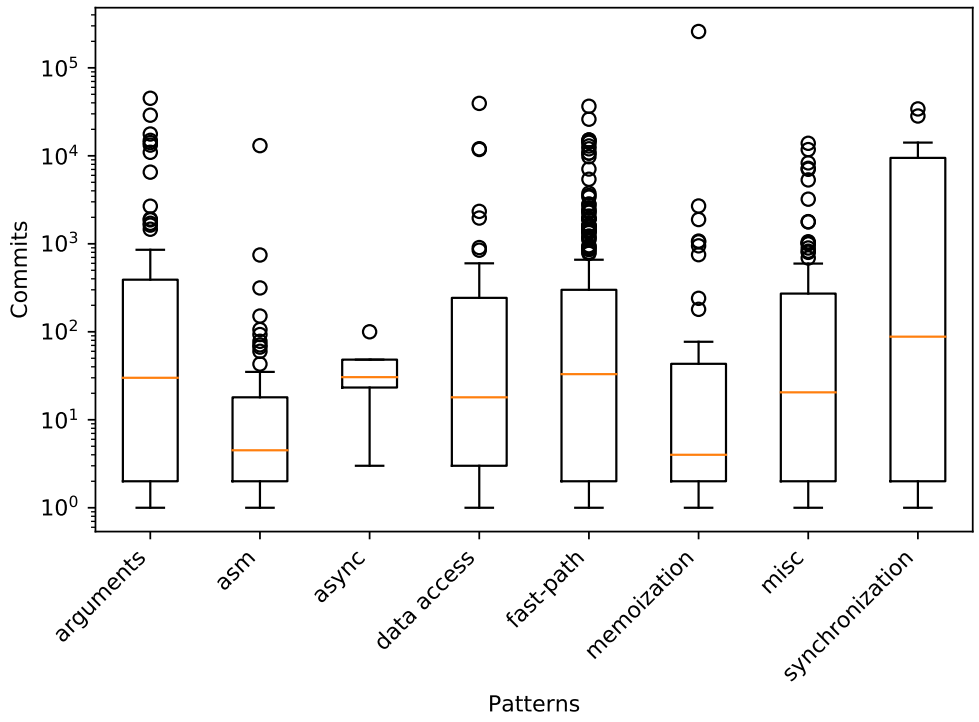


Fig. 3.3 Performance bug fix duration for different bug patterns measured by FTCF (see Section 3.2.3)

Synchronization related bugs have largest discrepancy in FTCF and a 75 % quartile that is two orders of magnitude higher than the median. In Section 3.3.5 I observed that modern synchronization optimizations often either adapt the RCU mechanism or alter the synchronization across concurrent threads. Substituting existing reader-writer locks with RCU is likely to require little effort, while reasoning about and fixing an inefficient synchronization without introducing a race is likely to require more time.

Another observation is that both the fast-path and arguments patterns have a high number of outliers. On the one hand this shows that the majority of performance bugs can be fixed very fast, on the other hand a small amount of such bugs need significantly more time to fix. This observation indicates that my data set comprises few difficult cases of these classes that appear to be challenging in addition to the larger group of simpler cases.

3.4.3 Performance Bug Fixing Developer Experience

The second metric indicating performance bug difficulty (*seniority* in Section 3.2.4) reflects the experience of developers who fix performance bugs. The baseline of the seniority metric is 0, when time between the first commit of the bug fixing developer and the bug fixing commit is the median across the respective time differences for all candidate developers who could have fixed the bug at that time. Figure 3.4 shows the seniority of fixers for each of the patterns. All boxes in Figure 3.4 have the median greater than 0, which means that performance bug fixing developers usually fall into the group of more experienced developers. Few boxes cross the 0 mark, indicating that for most projects less than 25 % of the performance bug fixes are contributed by the 50 % of the developers who have most recently joined the project. This indicates that fixing performance bugs is likely to require a certain degree of familiarity with the project code and that existing performance diagnosis tools may be difficult to use for less experienced developers.

3.4.4 Performance Bug Fix Size

The number of lines that a bug fixing commit consists of is another measure how complex the performance bug fix is. Figure 3.5 shows the number of modified lines across the patterns[9]. Although different patterns yield different complexity in terms of the code changed, the most frequent patterns have relatively low numbers of changed code lines. Complex code changes comprising hundreds of lines, such as for asm and async, are either project specific or less common. This essentially means that performance bugs can be generally fixed by touching a relatively small amount of source code.

[9]Estimated by `diffstat -m`

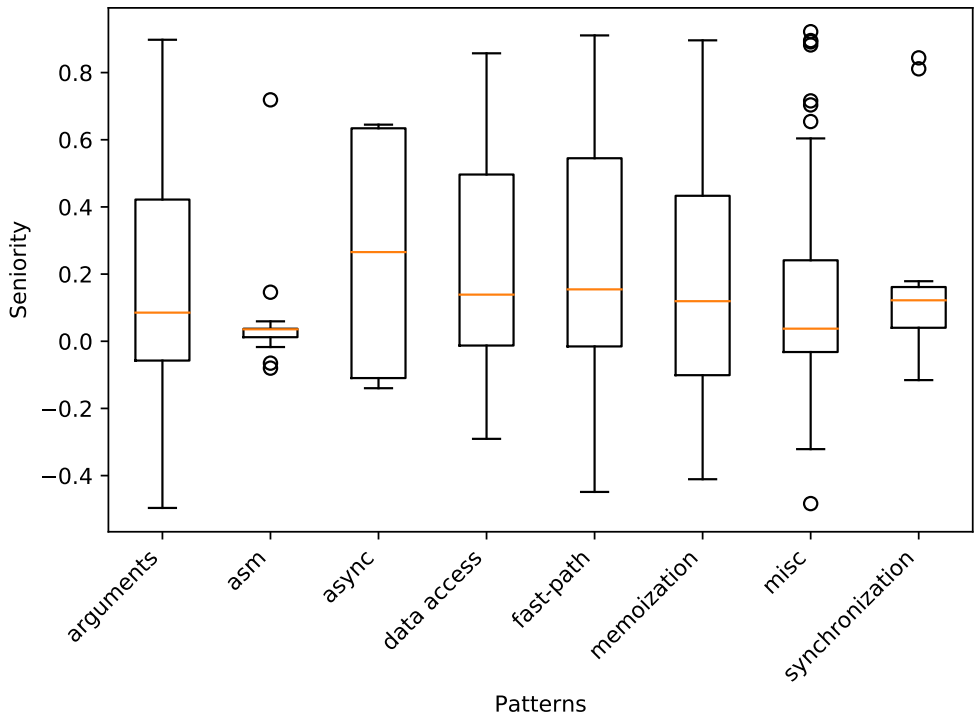


Fig. 3.4 Seniority of bug fixing developers across performance bug patterns. The metric captures the distance of a project-local seniority metric from the median seniority of all candidate developers for the fix on the same project. A seniority of 0 indicates experience matching the median, positive seniority higher experience, and negative seniority lower experience.

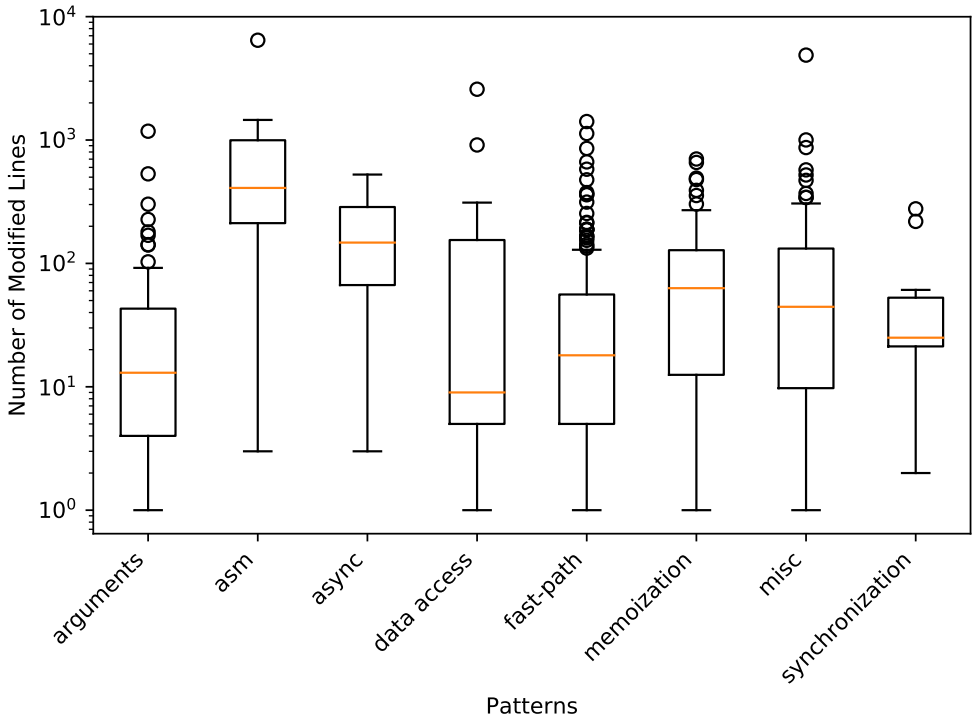


Fig. 3.5 Modified source lines of code per bug fixing commit across performance bug patterns

Since I envision performance mutations to support future work on performance bug detection, localization, and repair, it is also meaningful to study in which form performance bugs are fixed to guide the creation of corresponding mutation operators. Figure 3.6 shows the type of changes across performance bug fixes. The most dominant code change type to fix performance bugs is the addition of source code lines. Consequently, mutation operators resembling the identified performance bugs should mostly focus on code removal. This finding is not surprising, as fast-path implementations usually entail the addition of logic to identify the condition under which the fast-path can be executed and the actual implementation of the faster operation. Unfortunately, generating and adding semantic-preserving code must be expected to be simpler than ensuring that code removals are semantic-preserving, which I consider the main challenge for the realization of realistic performance mutations given the presented observations.

3.5 Threats to Validity

There are a number of threats to the validity of the conclusions presented in this chapter.

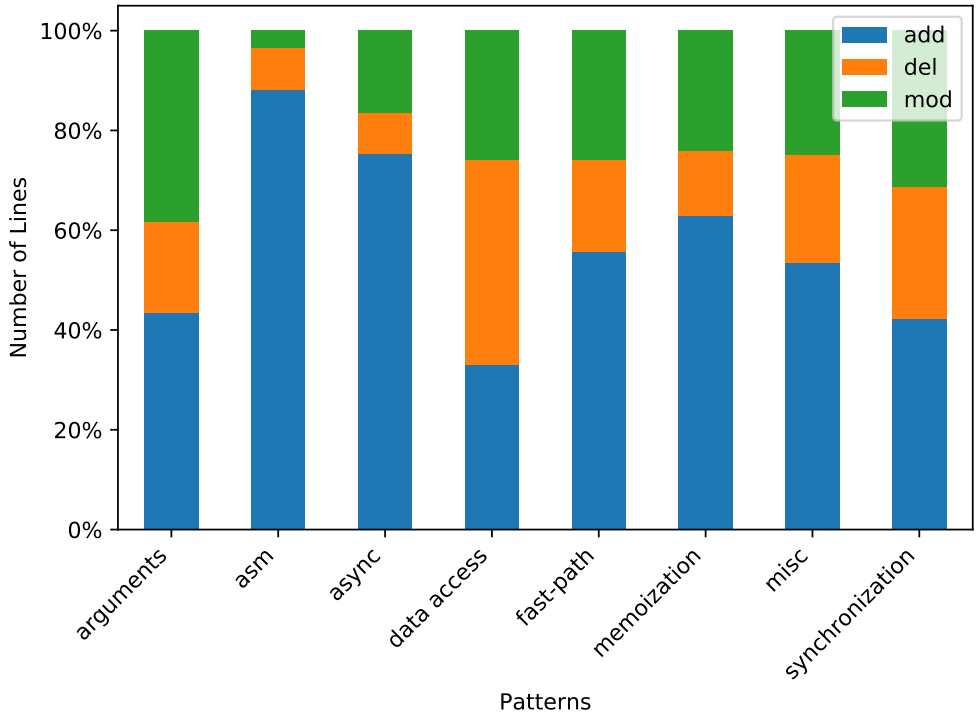


Fig. 3.6 Relative distribution of source line of code change types (addition, deletion, modification) in performance bug fixes by bug pattern

First, I apply a number of heuristics to infer various performance bug characteristics. My FTCF metric to approximate fix duration in a project-agnostic way requires knowledge of the bug introducing commit and this is based on the unverified assumption that in general each line of a bug fixing commit is necessary for the fix. This is a conservative assumption and, as a consequence, the exact fix times underlying Section 3.4.2 could in fact be longer and performance bugs more difficult to fix.Similarly, I approximate developer experience by the duration a developer has been active on the project before contributing the bug fixing commit. This may not accurately reflect developer experience if the developer has been actively developing other projects before. However, my manual investigation of the bug fixing commits show that fixing performance bugs often requires detailed project knowledge, which is less likely to be transferable from other prior projects.

Another threat lies in the criteria of the presented taxonomy, as the borders among semantic categories are fuzzy. In order to limit the impact of this threat I make my entire data set publicly available[10] for reuse and cross-validation.

The third threat comes from the fact that this study focuses on C projects with the exception of `llvm` and `clang`, because I assume that C is mostly used for "low level" programming for which performance is of a higher concern. In industry, C++ is also used for low latency applications where performance matters. While it is possible that different language features result in different syntactic manifestations of performance bugs, I do not expect this to significantly affect the presented results, as the presented taxonomy explicitly abstracts from syntactic details. Accordingly, the performance bug distribution of `llvm` and `clang` is similar to the C projects according to Figure 3.2. Therefore, the project coverage bias is unlikely to defy the results in Section 3.4.

The fourth threat is the incompleteness of the keywords list used to search for performance bugs. It is impossible to manually enumerate all relevant keywords with the guarantee of completeness. Sometimes certain functional bug fixing commits may lead to performance degradation, e.g. the commits to mitigate spectre introduced over 10% performance overheads [128].

The last threat is attributed to the selection of popular Debian packages ranked by votes, because there is no guarantee that the votes really reflect the popularity of these packages. Nevertheless, the number of projects investigated is large enough to compensate the drawbacks of possibly selecting less popular projects.

[10]https://yqchen.gitlab.io/perf-bugs/

3.6 Conclusion

In this study, I assess the alignment of current research and tool development in the area of performance bug detection and localization with actual performance bugs derived from 733 performance bug fixing commits across 13 open source projects written in C and C++. I manually investigated these commits to confirm they actually constitute performance bug fixes and to group them in semantic categories according to how they intend to achieve a speed-up of the modified code. In summary, I found that more than half of the studied performance bug fixes introduce fast-paths in the control flow or tweak arguments to trigger the execution of existing fast-paths.I define a set of three complexity metrics suited for cross-project comparison, which are related to bug fix duration, developer experience of bug fixing developers, and the amount of code changed by the fix. The empirical assessment of these metrics shows that usually 10 to 100 commits lie between the introduction and removal of performance bugs, that performance bugs tend to be fixed by more experienced developers, and that the lines of code that performance bug fixes comprise greatly vary across different bug categories. From these observations I conclude that performance bugs are fixed in a relatively short time for active projects, but that existing tool support does not effectively target less experienced developers, which is a strong motivation to develop more effective and intuitively usable tools. I also found that performance bug fixes usually entail 10 to 100 changed lines of code, most of which are code additions. Finally, my results provide important insights about the distribution of performance bugs in software projects and about the complexity of these bugs. Besides guiding work on performance bug detection and localization approaches, this data is valuable for the generation of realistic performance mutants to support a fault-based assessment of these approaches.

Chapter 4

Performance Mutation Testing

Performance bugs are unnecessarily inefficient code chunks in software codebases that cause prolonged execution times and degraded computational resource utilization. For performance bug diagnostics, tools that aid in the identification of said bugs, such as benchmarks and profilers, are commonly employed. However, due to factors such as insufficient workloads or ineffective benchmarks, software defects related to code inefficiencies are inherently difficult to diagnose. Hence, the capabilities of performance bug diagnostic tools are limited and performance bug instances may be missed. Traditional mutation testing (MT) is a technique for quantifying a test suite's ability to find functional bugs by mutating the code of the test subject. Similarly, I adopt performance mutation testing (PMT) to evaluate performance bug diagnostic tools and identify where improvements need to be made to a performance testing methodology. I carefully investigate the different performance bug fault models and how synthesized performance bugs based on these models can evaluate benchmarks and workload selection to help improve performance diagnostics. In this chapter, I present the design of a PMT framework, SLOWCOACH, and evaluate it with over 1600 mutants from 4 real-world software projects.

4.1 Introduction

A program's performance is a software attribute that describes how quickly it can complete tasks or process inputs [84]. Performance is important when input sizes increase but program throughput does not scale accordingly. One of the causes of scalability issues is the presence of unnecessarily inefficient code within a program's codebase, which wastes computational resources when executed. For example, some function may not save the frequently needed result of an expensive computation, but instead recomputes it each time when needed, thereby wasting CPU cycles. Or, in a lock contention scenario, inefficient synchronization

code may causes a program to wait unnecessarily for off-CPU events. Inefficient code chunks that could be optimized to increase program performance are often referred to as *performance bugs* [116, 144, 9]. The slowdowns caused by such unoptimized code parts can be measured subjectively, e.g., users report some programs to be slower than their expectation, or objectively, i.e., measuring and comparing performance metrics, usually wallclock time.

An essential part of code optimization is to identify the inefficient code. The identification is usually carried out in two steps: 1) *detection*, i.e., determining whether performance issues exist in the first place, and 2) *localization*, i.e., pinpointing the code chunks causing the issues. Several approaches and diagnostic tools exist that assist in the detection or localization of performance bugs to guide performance diagnostics and optimization [145, 148, 144, 124, 152]. Some syntax checkers [84], for example, find simple performance anti-patterns in code statically and thus help to avoid them. However, most performance bugs are too complicated for simple syntax rules to detect and must be analyzed with runtime information [132, 144, 168]. Given suitable workloads, benchmarks provide performance metrics that can be used as a comparison basis, while profilers provide runtime information that helps developers localize performance bugs. Still, neither benchmarks nor profilers can ascertain whether there are performance bugs without a proper specification or performance measurements from previous versions for comparison [25].

More sophisticated approaches [145, 20] aim to detect performance bugs by symptomatic analysis, e.g., tracing hardware/software events or memory accesses. Such approaches, however, lack a ground truth to be evaluated against. The symptoms on which they depend are not guaranteed to have observable performance degradation. This observability problems can be caused by many factors, such as insufficient workloads. For example, a vector in C++ reallocates memory when new elements are added and no memory is available to hold them. Such reallocation has a very small performance overhead, making it challenging to measure with profilers. But reallocation would cause significant performance degradation if it occurs often [132]. Thus, it is not always clear that the symptoms identified by such approaches are actually performance bugs, as the given workload may not be able to exercise the problematic code frequently enough.

Moreover, there is no simple way to evaluate performance diagnostic approaches beyond the small set of a priori known and reproducible performance bugs [20, 164, 168]. The lack of a standardized *corpus* of performance bugs and the lack of rules for synthetically creating performance bug instances to evaluate performance bug detection and localization approaches, motivate my interests in the synthesis of performance bugs. Moreover, synthesized performance bug instances help reproduce many historical performance bugs that cannot be reproduced due to dependency on outdated system configurations.

Inspired by the idea of software fault injection, I adopt techniques from *mutation testing* (MT) [123, 78] to inject performance bugs to evaluate the quality of existing performance bug detection and localization approaches. MT intentionally injects synthetic faults, using code mutation, into the test subject's code to check if its test suite can find them. The ultimate goal is to quantify and improve the quality of test suites. The rules controlling how and where the source code is mutated are known as *fault models*. After mutation, the source code is expected to produce functional deviations compared to the original code. A high-quality test suite is supposed to capture these deviations. In this chapter, I employ *performance mutation testing* (PMT) to synthetically create performance bugs as an assessment for the performance testing. In stark contrast to MT, PMT requires that code mutations *do not* introduce functional deviations since performance bugs should be considered separately from functional bugs. Therefore, PMT requires the preservation of functional equivalence (FE). This makes PMT different from traditional MT techniques, which have the exact opposite requirement.

In this chapter, I introduce SLOWCOACH, a novel PMT framework. I demonstrate its practical utility and show how PMT can help to improve performance bug diagnosis approaches. Particularly, I address the following research questions in this chapter:

1. How does PMT relate to MT?
2. What are the different dimensions to consider for PMT fault models?
3. Can PMT produce enough useful mutants in practice?
4. How useful are synthetic performance bugs in practice?

4.2 Background

4.2.1 Performance Mutation Testing

Mutation testing (MT) is a technique to evaluate test suite quality [78, 123]. In MT, faults are injected through code mutations, commonly at the source code level. The code transformation rules that govern how to mutate the original source code are called *mutation operators*. The resulting copies of mutated code are termed *mutants*. A *fault model* describes what, where, when, and how to inject the buggy code. An example MT fault model could match all binary *and* operators (&&) in `if` statements and change them to binary *or* operators (||). If a software's test suite is not able to distinguish the generated mutants from the original software, then the test suite needs improvement as it fails to detect the injected faults. I compare MT and PMT in Table 4.1 and provide further definitions of MT concepts.

In the context of performance evaluation, a test suite is typically represented by benchmarks and workloads, which exercise the software under test with input data to reveal performance regressions and to diagnose bottlenecks. Similar to MT evaluating test suites, I adopt PMT as technique to evaluate performance benchmarks and performance diagnostics approaches. However, MT fault models aim to change the functional behavior of the original code, since these are the faults that test suites should detect. In practice, mutants that *do not* change functional behavior (equivalent mutants) are avoided or filtered out if possible. In contrast, PMT filters out mutants that *do* change functional behavior as it is only meaningful to compare the performance of functionally identical programs. Thus, an important requirement for PMT fault models is to retain the functional behavior for all mutants while introducing performance overheads. These concepts are detailed in Table 4.1. As such, P-mutants are different from traditional mutants in terms of key MT concepts. To better distinguish these mutant types, I refer to such performance mutants as *P-mutants*.

The FE of P-mutants is defined as: *given a set of inputs, all P-mutants should produce the same set of outputs as the unmutated code*, i.e., all P-mutants should adhere to the same functional specification as the original code. Despite much proposed work addressing the program equivalence problem in the research community (e.g., [65, 96]), the preservation of FE in PMT cannot be trivially solved. The proposed formal equivalence checkers verify if two programs execute the same steps, while performance optimization in general involves two versions of a program that produce the same output by executing correspondingly fewer steps. Conversely, PMT fault models would lead to more steps being executed in a program. So, formal FE checkers cannot be used to effectively verify FE for PMT. An alternative approach proposed by Devroey et al. [53] detects equivalent P-mutants by the simulation of non-deterministic automata. However, none of these approaches can generally solve the FE problem in acceptable time for potentially thousands of mutants. As a practical compromise, I carefully select PMT mutation operators that are unlikely to affect functional behavior and check FE using classical functional tests.

4.2.2 PMT Fault Models

Performance bugs are often believed to be fixed by "relatively simple source code changes" [84]. However, they usually involve more complicated semantic changes in the real world [27, 134].

Jin et al. [84] identified performance bugs via detectors which relied on the *contextual information* of the code. For example, given function A invoked before function B would cause performance degradation, a detector finds all invocation pairs of function A before function B. The contextual information in this example is the relative invocation ordering of functions A and B. Some PMT fault models inject performance antipatterns derived from

Table 4.1 Performance Mutation Testing vs. Mutation Testing [112]

Concept	Performance Mutation Testing	Mutation Testing
Test Suite	A fixed set of benchmarks and workloads yielding various performance metrics (e.g., execution time, memory usage, and execution paths) to be compared against. The workload carried out by the benchmark aims to identify those tests that have worse performance metrics than the unmutated baseline.	A test suite (test programs and inputs) to determine whether a program complies with its (functional) specification. Tests should *kill* (i.e., detect) mutants to demonstrate their fault detection capability.
Equivalent Mutants	All performance mutants must be functionally equivalent to the original version. Performance equivalent mutants are those whose performance results are statistically close to the original.	If a mutant is functionally equivalent to the original software, this mutant will never be killed by the test suite.
Hard-to-kill Mutants	Functionally equivalent mutants that can be killed only if mutated code is executed sufficiently frequently. I hypothesize that all code changes introduce performance overheads if not optimized out, while the overheads need repetitive execution to be observable.	Some mutants can only be killed by few, very specific test cases. These mutants help identify possible improvements of test suites.
Mutation Score	Identical to the mutation score by mutation testing. But since there are multiple performance metrics, there are correspondingly multiple mutation scores for the different perspectives of the metrics.	The mutation score grades the quality of a test suite. It is the percentage of nonequivalent mutants that can be killed by the test suite.

the detection strategies, which also requires domain knowledge. As an example, the code shown in Listing 5 demonstrates a performance optimization scenario in the real world [27]. The code snippet matches a string against a pattern[1], this algorithm uses Deterministic Finite Automata (DFA) or keywords searching to perform the matching. DFA usually matches patterns with wildcards faster than the keywords searching algorithm, if the inputs are unibyte and do not contain any back references. Hence in Listing 5 I search by the DFA if the condition `dfafast` is satisfied (line 9 and 19). This example will be the first case study discussed in Section 4.4.4.

The fault models derived from this example could be either to remove the `else if` block or to change the value of `dfafast`. Unfortunately, neither model can be described by simple syntactic rules without contextual knowledge about what the affected code blocks or variables are used for. Such fault models suffer from several drawbacks. Firstly, large human-in-the-loop efforts are required to understand the entire software project and to implement

[1]The code is simplified for the discussion.

```
+bool dfaisfast (struct dfa *d) {
+   return !d->multibyte &&
+       d->has_no_backref();
+}

 size_t EGexecute (char const *buf,
        size_t size, size_t *match_size,
        char const *start_ptr) {
+  bool dfafast = dfaisfast (dfa);
   /* ... */
   for (beg = end = buf; end < buflim; beg = end){
       if (!start_ptr) {
           if (kwset) { /* Slow path */
                 do_kwset_search();
                 /* ... */
                 if (matched) return;
             }
-          else
+          if (!kwset || dfafast) {
               /* Fast path */
               do_dfa();
```

Listing 5 Fast Path

these mutation operators. Secondly, these fault models generate only a limited number of P-mutants. In my experiments on `grep` (discussed in Section 4.4.4), each mutation operator instance[2] generates about 1 to 2 P-mutants. PMT fault models that do not rely on domain knowledge are more generic than their counterparts that use contextual information. I classify the space of possible PMT fault models along two dimensions: how representative and how context dependent fault models are. Representativeness can be categorized as the fault models simulating performance bug effects and developers' errors. The other dimension specifies if a fault model is context dependent or independent. This is visualized in Figure 4.1. Since the Listing 5 simulates developer errors, the described fault models fall into quadrant 2, or $\mathbb{Q}2$ in short, as they depend on contextual information. The context-independent fault models fall into $\mathbb{Q}4$ as they do not rely on contextual information when injecting faults. These two families of fault models inject performance bugs by simulating developer errors, but differ in their generality as indicated on the x-axis in Figure 4.1.

[2] SlowCoach embeds the contextual information into mutation operators. Each operator with the contextual information is an instance. (c.f. Section 4.3.1)

An alternative to the simulation of developer errors is to simulate the effects of performance bugs. All performance bugs have a performance impact either on-CPU or off-CPU. For example, an on-CPU performance issue can be caused by inefficient algorithms that waste CPU cycles, and an off-CPU issue can be related to unnecessary file IO operations that cause wait times. The on-CPU overheads can be simulated by inserting useless operations into the code, while off-CPU overheads can be emulated by inserting useless `sleep()` operations. Although these simulations do not hamper the FE and potentially generate more P-mutants, the introduced code mutations do not resemble performance bugs found in real world software. The resemblance of synthesized bugs to those that occur in the real world is called the *representativeness* of software faults [111]. The simulation of performance bug impacts is less representative when compared to the simulation of developer errors as indicated on the y-axis in Figure 4.1. The representativeness of MT fault models is a significant indicator for the efficacy of MT. Performance bug detection and localization approaches, however, are only concerned with the symptoms of performance bugs. In other words, a performance bug may itself be trivial, but it is not trivial to evaluate whether a particular benchmark or workload is capable of showing observable performance degradation.

Like other PMT fault models, the effect simulation may also be dependent on domain knowledge. For example, as there are no consistent interfaces among C/C++ software projects for low-level system operations, the function names of these operations are required for fault models. In the case of heap memory, allocations are performed with `malloc()` in standard C and with `new/new[]` in C++, but many projects adopt custom allocators, e.g., `kmalloc()` in the Linux kernel, and `ALLOC()` or `xmalloc()` in gnulib. I label the simulation of performance bug effects as $\mathbb{Q}1$ and $\mathbb{Q}3$ in Figure 4.1 with and without context dependency correspondingly. In spite of the context dependency, the difference between $\mathbb{Q}1$ and $\mathbb{Q}3$ is often negligible in terms of simulating performance bug causalities, which is why I usually discuss $\mathbb{Q}1$ and $\mathbb{Q}3$ fault models together. As SLOWCOACH allows developers to encode contextual information in the fault models, this means there are limited differences between $\mathbb{Q}1$ and $\mathbb{Q}3$. Therefore, I use $\mathbb{Q}3$ to represent $\mathbb{Q}1/3$ mutants in the following discussions.

As discussed in Section 1.3, the coupling effect [52, 117] is one of the two basic assumptions by mutation testing, which states that by addressing a bug, similar bugs in the same category can be addressed easily as well. In the context of PMT fault models, quadrants in Figure 4.1 show a large discrepancy between categories in terms of the applicability of coupling effects. The coupling effects by $\mathbb{Q}2$ and $\mathbb{Q}4$ remain similar to those by tradional mutation operators, as each operator of $\mathbb{Q}2$ and $\mathbb{Q}4$ assimilates a type of performance bugs in the real-world. For $\mathbb{Q}3$ operators, coupling effects see limited application, as the causal-

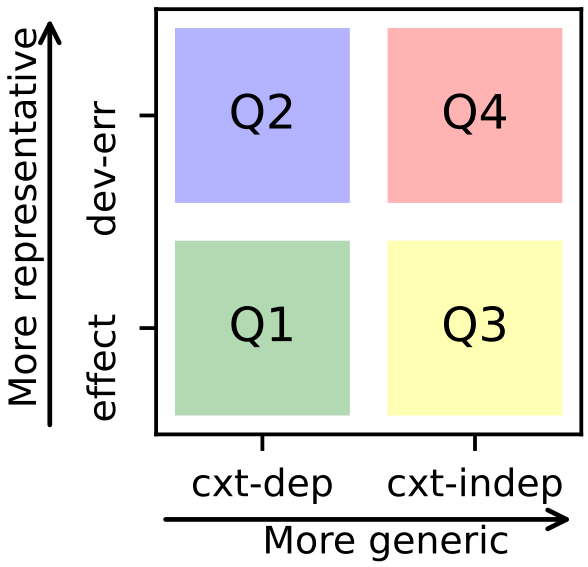


Fig. 4.1 PMT Fault Models

ity of performance bugs may differ from the injected faults, e.g., we could use inefficient loops [144] to simulate the performance impact of spinlock contentions [102, 15].

4.3 SlowCoach: A PMT Framework

4.3.1 Overview and Workflow

A general overview of SlowCoach's workflow is provided in Figure 4.2. SlowCoach's primary inputs are the source code of the target software project and project specific configuration which must be provided by the user. The configuration allows customizing the PMT process for a specific project, such as function ignore or include lists, required for some mutations operators to work correctly. Section 4.3.1 shows a configuration example to replace all local variables in the `main` function named `dfafast` (in Listing 5) with a value of `false`. One or more optional `caller` elements can be provided to limit the scope of replace operations to a specific set of functions. The tool mutates the original source code to generate different mutants (as modified source code files), which are then applied to unique copies of the project. Both the original project and the mutated versions are then compiled to generate the executable programs. The programs are then executed with benchmarks using various workloads, or other performance diagnostic approaches are applied to the executables. The performance metrics the user is interested in are measured and recorded for the original and all mutated versions. Based on these metrics or their comparison across versions, the user can assess the quality of the used benchmarks, workloads, or performance diagnostic tools.

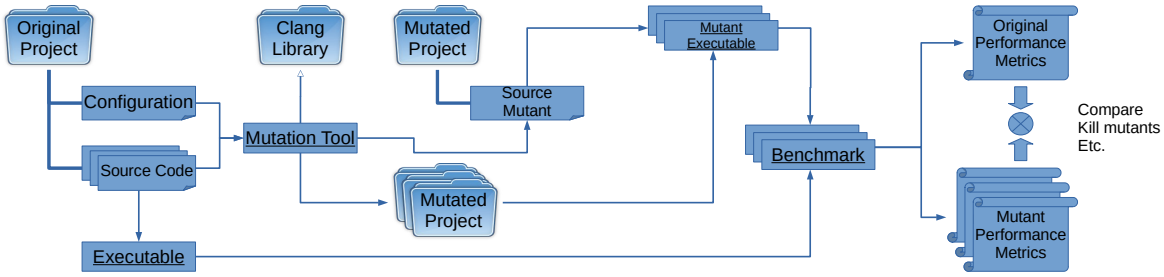


Fig. 4.2 SLOWCOACH Workflow

```
<local-var>
  <var-name>dfafast</var-name>
  <value>false</value>
  <caller>main</caller>
</local-var>
```

Listing 6 Configuration Example

4.3.2 Mutation Operators

In this section, I take the operators from Table 4.3 in Section 4.2 to illustrate how PMT mutates the code.

$\mathbb{Q}4$ Operators (developer errors, context-independent)

$\mathbb{Q}4$-A – Loop Unbreaker As discussed in Section 4.2, the fault models simulate developer errors without contextual information. For SLOWCOACH, I select 2 representative mutation operators to demonstrate the concepts of $\mathbb{Q}4$ fault models and discuss two of them in this section. The first operator is called *loop unbreaker* and derives from a common performance optimization pattern found in many real-world projects [144, 27] that I call *loop breaker*. This pattern is similar to the fast path pattern in Listing 5. I consider that early termination (`break`) of a loop is a fast path, stopping the loop early when possible. Section 4.3.2 shows an example for the loop breaker pattern. The `if` statement containing the `break` on line 2 in Section 4.3.2 is the fast path that terminates the loop early. The loop *un*breaker mutation operator removes these `if` statements. Though $\mathbb{Q}4$-A shares some similarity with the **goto label** operator in classic mutation testing [174], this operator concerns more on functionality preservation and performance degradation within the strucutred programming scheme [54].

The main drawback of loop unbreaker is that semantics cannot be asserted from the syntactic `if` statement. It is possible that the `if` break is necessary for functional behavior, e.g., a loop returning the first occurrence of an item in a list. Moreover, loops may rely

```
for (int i = 0; i < 1024; i++) {
    if (some_cond(i)) break;
    do_something();
}
```

Listing 7 Loop Breaker optimization pattern (ℚ4-A in Table 4.3)

on such an `if` break to terminate, so the removal of it may cause the program to hang. Although loop hangs can be detected by runtime monitoring [39, 175], it is hard or even impossible to predict statically during code mutation. To minimize the probability that the loop unbreaker operator causes program hangs, I adopt a naive heuristic to exclude loops without a terminating condition, e.g., `for(;;)` or `while(1)`, because these loops rely on a `break` statement to terminate.

ℚ4-**B – Oblivion** The second mutation operator is called *oblivion*. It is derived from another performance optimization pattern called *cache memoization* [51].In this pattern the code records the result of some heavy computation and reuses it later without re-doing the computation again. A simple example is shown in Section 4.3.2 at line 1 to 2 in Section 4.3.2, where the results of `foo()` are cached in variable `a` and re-used in `bar(a)`. To simulate performance bugs where cache memoization was omitted, the oblivion operator substitutes all variable references with their initializers, so that the result of the initializing function is redundantly computed. In Section 4.3.2, the oblivion operator matches all occurrences of variable `a` and substitutes them with a function call to its initializing function `foo()` as shown on Section 4.3.2 line 3.

Fig. 4.3 Cache Memoization optimization pattern (ℚ4-B in Table 4.3)

```
int a = foo();
bar(a);
bar(foo());
bar((a == foo()) ? foo() : a);
```

The oblivion operator, however, may alter the functional behavior if the relevant function updates or depends on the global state of the program, which is similar to the **statement deletion** operator in traditional mutation testing [174]. A notorious example is memory allocation, whose results depend on the global state and repeated calls to memory allocation functions lead to different results. To mitigate this, I adopt a straightforward function

blacklisting approach, i.e., the code is mutated only if a local variable is declared with an initializer and the initializing function is not blacklisted. As a result, given a presumably side-effect free initializer, the oblivion operator adds a ternary operator as shown line 4 in Section 4.3.2, where the local variable `a` is replaced with an expression of the form: `(a == foo()) ? foo() : a`. If the initializer function (`foo()`) returns a different value, variable `a` was changed since its initialization and I fall back to using variable `a` directly to not affect the original program semantics. While this approach amplifies the performance impact by calling `foo()` twice in the ternary operator, it increases the probability that the original functional behavior is retained if `foo()` is side-effect free. Despite these efforts, it is still possible that the functional behavior is changed, for example, if the initializer function is not side-effect free or takes arguments that change over time.

$\mathbb{Q}2$ Operators (developer errors, context-dependent)

The performance bug dataset provided by Chen et al. [27] shows that almost all real-world performance bugs and optimizations depend on project specific context, which has been confirmed by other works on real-world performance bugs [144, 134].Therefore, additional contextual information is usually needed to synthesize representative performance bugs since semantic fault injection is not possible with purely syntactic rules in most MT approaches [123, 78, 21]. SLOWCOACH encodes the contextual information in the configuration as described in Section 4.3.1 and shown in Figure 4.2. Considering the fast path example in Listing 5, occurrences of the `dfafast` variable can be replaced with a `false` so that the fast path will never be taken. A mutation operator that removes fast paths can then match the variable reference with this name, its parent `if` statement, and its enclosing function `foo()`, and safely remove this particular fast path as defined by the user.

To demonstrate how context dependency is encoded into PMT operators, I take three examples of code optimizations from the GNU `grep` project (one of the most popular text search utilities) and show how they can be reversed to produce performance bugs. The Git commit ID in the official GNU `grep` repository [151] is provided for each example. Detailed code changes and performance bug reproduction can be found at Appendix A.

In the first case[3], developers introduce a new function `dfaisfast()` (described in Listing 5) as the switch between the fast path and the slow path. In `grep`, both a DFA (deterministic finite automaton) and a keyword trie are implemented to match regular expressions (regex) [2, 32]. Using the DFA is usually faster than keyword trie matching if keywords (patterns) are unibyte and there is no back reference in the regex. So the `grep` developers added a new function `dfaisfast()` to check these two conditions. If both conditions are

[3]GNU `grep` repository [151] Git commit ID 3255bc

met, DFA matching is used, and trie matching otherwise. The check is performed early and its result is cached in a variable called `dfafast`. This is similar to the example in Listing 5 in that `dfafast` corresponds to `some_cond` and the trie algorithm corresponds to `heavy_computation()`. SLOWCOACH provides a mutation operator that replaces a call to a given function with some other function, a variable, or a concrete value. This operator can replace the call to `dfaisfast()` with a fixed value of `0` to change the fast path condition, thereby permanently preventing the fast path from being executed. For both cases, ℚ2-A and ℚ2-B operators Table 4.3 are instantiated to replace occurrences of all calls to `dfaisfast()` and `fgrep_icase_available()` with a value of `0`. This case is detailed in Section A.1.

In the second case[4], `fgrep_icase_available()` is added to the code, which functions like `some_cond` as well. The `fgrep` variant of `grep` has a dedicated `fgrep` engine for fixed string searches, but it still relies on the efficient `grep` engine as the backend in certain cases. Although the `fgrep` engine is faster than the `grep` engine if only unibyte characters are involved in a case insensitive search in multibyte locales, the `grep` engines was originally used for such searches. As in the first case, developers introduced a new function `fgrep_icase_available()` that can identify these cases. The same mutation operator as before can be used to replace the calls to `fgrep_icase_available()` with a value of `false` to prevent the fast path from ever being taken. Section A.2 contains the code changes of this case.

The third case[5] is called "argument" by Chen et al. [27], which is to avoid heavy computation by controlling the arguments passed into this computation. Here, code is re-arranged in `grep`'s PCRE matching engine to speed up the skipping of encoding errors. Here, the PCRE engine is slow to process input texts containing multibyte locale encoding errors. The variable `subject` represents the input text (index of the input text) to be passed to the PCRE engine. So, developers update `subject` to skip encoding errors. A variable `p` is used as iterator over the input text and variable `subject` points to the next valid input text to be passed into the matching engine. If there are encoding errors in the text, `p` will skip over them but leave `subject` pointing to the error containing text. Hence, the matching engine must handle these errors, which is relatively slow. As optimization, developers re-arranged the code to update `subject` with `p` to avoid the slower error skipping in the matching engine. The code change by this case is in Section A.3.

ℚ1 & ℚ3 Operators (performance effects, context-dependent and -independent)

[4]GNU `grep` repository [151] Git commit ID 960ad3
[5]GNU `grep` repository [151] Git commit ID 5cb49d

```
volatile int sum = 0, foo[ARR_LEN];
for(int i = 0; i < foo_len; i++) {
    sum += foo[i];
}
```

Listing 8 **1*** Loop (ℚ3). Produces useless results in each iteration.

As discussed earlier, mutating for performance while retaining the functional behavior of programs is generally a hard problem. ℚ4 operators require expensive static or dynamic analysis if unchanged functional behavior must be guaranteed. ℚ2 operators compromise on generality for a more accurate reincarnation of previously known performance bugs. Since most real-world performance bug fixes involve specific inputs, the lack of generality limits the usefulness of ℚ2 operators for evaluating larger sets of workloads. An alternative to ℚ4 and ℚ2 operators is to simulate the observable effects of performance bugs rather than the bugs themselves. A naive mutation operator could insert `sleep()` operations into the code, to extend the wall clock time of the execution. But inserting `sleep()` does not aid in improving performance benchmark design workload selection, because sleeping does not affect the CPU time, hence can be easily detected by checking the CPU utilization.

Loops are often considered as one of the main sources of performance bottlenecks [144, 116, 92, 126], but reversing loop-related performance optimizations may introduce functional behavior deviation. To simulate the effects of loop-related performance bugs, I develop mutation operators to synthesize inefficient loops. I derive the ℚ3 fault models from the inefficient loops classified by Song and Lu [144]. There are many types of inefficient loops, e.g. **1***, **0*1?** or **0*1?**. Since I are simulating the effects of performance bugs by ℚ3 operators, I do not discuss all types of loops in detail. I pick a typical inefficient loop known as the **1*** loop for the PMT study and evaluation. **1*** loops produce results (side effects) in each iteration, where these results are useless. A simplified example is the loop in Listing 8 which computes the sum of an integer array. Since the variable `sum` is written by the incremented value of `foo[i]`, there is a result in every loop iteration. However, these results (accumulated as `sum`) are not used after the loop, i.e., they are unnecessarily computed. Although most **1*** loops are semantically related to the loop context and much more complicated, ℚ3 mutation operators could use a simple form (e.g., summing integer) to simulate the performance bug effects. Other loops like **0*1?** and **[0|1]*** with different memory access patterns can be easily implemented and encoded in SLOWCOACH. Due to high similarity in terms of introducing performance impacts for PMT, I only evaluate the mutation operators derived from **1*** loops in this chapter, and apply various array lengths (`ARR_LEN`) to simulate different

performance impacts. The `foo` array is allocated on stack to avoid randomness caused by dynamic allocators (line 1 in Listing 8). Since static arrays will be optimized by compilers, `volatile` was used on `foo` to prevent the compiler removing injected loops.

4.3.3 Implementation

In this chapter, we developed a PMT framework prototype named SLOWCOACH with the general workflow in Figure 4.2. The dashed lines in the figure represent the dependency of classes, where the arrow means the source depends on the target. The line with arrow indicates the inheritance. We take source codes in a project and a configuration file in xml format and specific to this project as the input of my PMT framework. The configuration file comprises a list of operators to be instantiated. Each operator derives from `clang` syntactic matchers to locate point of interests (POIs) and implements a set of heuristics supported by contextual information to mutate the code. As an example in Listing 5, we want to remove the occurrence of `if` statement, but only in the function bodies of `foo()`. This operator in SLOWCOACH is hence encoded as a configuration entry that matches `if` statements, along with the whitelist function `foo()` as its subnode. The singleton object `OperatorManager` [58] functions as a driver that initiates `Operator`s by the configuration and starts the matching process. The clang matcher is a "visitor" [58] binding POI and user defined operations by `run()` function in `Operator`, whose handler is implemented by its subtypes like `FooOperator`. The handlers generate `Mutant`s which contain the mutated source code, and is responsible to write the buffered code into files. These mutated source code files are the generated faults that can be injected into project copies.

4.3.4 Prototype Limitations

The SLOWCOACH prototype implementation is for research purposes and as such has limitations. It is built upon the `clang` compiler frontend and, therefore, only supports the languages supported by `clang`, namely C, C++, and potentially Object-C (as Object-C shares the frontend with C/C++ in `clang`). However, this is not a general limitation of the approach as other compiler frontends or parsers can be used to port the SLOWCOACH mutation engine to other programming languages such as Python [157]. The current prototype is limited to generating first order mutants only, i.e., one mutation operator can be applied one time to produce a mutant that contains exactly one mutation. This limitation will be lifted in the future with further engineering efforts invested into the prototype. We currently generate all mutants sequentially. With further engineering efforts, the mutant generation process can be expedited by employing parallelization. Also, as already discussed in previous operators, pure

Table 4.2 Evaluation Software Projects

Project	Application Area	PL	LoC	Mutants	T_m*	OH†
`astar`	Path-finding Algorithms	C++	3959	514	0.77	36.41
`bzip2`	Compression	C	7292	892	0.40	17.58
`mcf`	Combinatorial Optim.	C	2044	239	0.40	19.36
`grep` ‡	GNU Text Utility	C	357 520	1532		

* Time in seconds to generate all source code P-mutants.
† Total time in seconds to generate, sample, compile sampled P-mutants.
‡ `grep` has 1532 $\mathbb{Q}3$ and $\mathbb{Q}4$ operators and 3 extra case study P-mutants. Since `grep` only has a default functional test suite, which does not apply to performance testing as those by `SPEC`, the $\mathbb{Q}3$ and $\mathbb{Q}3$ P-mutants in `grep` are not evaluated and their overheads are hence not presented.

syntactic rules are not powerful enough to simulate developer errors regarding performance bugs. In particular for $\mathbb{Q}2$ and $\mathbb{Q}4$ operators, the syntactic representation of performance bugs could be uneunmerable regardless how semantically simple the performance bugs are. Moreover, these semantics must be inferred manually by developers, which introduces extra human efforts and potential false positives or negatives for PMT.

4.4 Evaluation

In this section we evaluate SLOWCOACH by applying it to 4 real-word software projects. The evaluation is driven by the following research questions.

RQ 1 What is SLOWCOACH's runtime overhead and how many P-mutants does it generate?

RQ 2 Which fraction of the generated P-mutants preserve the functional equivalence to the original version?

RQ 3 Does PMT assist in identifying issues with performance testing tools and the testing environment?

4.4.1 Experimental Setup

Implementation & Mutation Operators

We use the SLOWCOACH prototype implementation to conduct the experiments for this evaluation. The prototype targets C and C++ software as performance critical software is often implemented with these languages. The prototype itself is realized using C++ and Python. For its code mutation functionality, it builds upon the Clang C/C++ frontend in

Table 4.3 Mutation Operators.

Operator	Description	MPO_{O0}*	MPO_{O3}
ℚ2-A	Replace `dfaisfast()` calls with `0`		
ℚ2-B	Replace `fgrep_icase_avail()` calls with `0`		
ℚ3-A	Prepend `sleep(1)` statement to loop bodies	1000000	1000000
ℚ3-B	Prepend **1*** loop (10 000 iterations) to loop bodies	51.47	4.82
ℚ3-C	Prepend **1*** loop (100 000 iterations) to loop bodies	514.55	48.25
ℚ3-D	Prepend **1*** loop (1 000 000 iterations) to loop bodies	5137.72	479.16
ℚ4-A	Apply *loop unbreaker*, remove early `break` from loops		
ℚ4-B	Apply *Oblivion*, remove cache memoization		

* Time measured by Google microbenchmark, in microseconds.

version 10.0.1. The currently supported mutation operators (cf. Section 4.3.2) that we apply in the experiments are described in Table 4.3. Note that the ℚ2 operators are only used in the `grep` case study in Section 4.4.4 due to their program specificity and the manual effort involved.

Evaluation Targets

Since the prototype targets C and C++ software, we select evaluation targets implemented in these languages. Additionally, the selected targets should be sensitive to performance issues, i.e., good performance should matter to their users. For example, the performance of a compression program such as `bzip2` is important to its users as excessive processing time wastes resources. Targets should also be selected from diverse application domains to avoid domain specific bias. Considering these aspects, we choose the SPEC CPU 2006 benchmark suites [70] as the primary source of evaluation targets. Table 4.2 provides an overview of the selected target programs along with a size estimation as number of lines of code (LoC). The target programs for the case study are `astar`, `bzip2`, and `mcf` from SPECint 2006, as well as `grep`, which is a well known real-world utility.

Workloads

To exercise the evaluation targets, we use workloads of different sizes. For the programs from SPEC, we use the standard workloads (inputs) that come with SPEC. For `grep`, we use the developer test suite.

Experiment Execution

We generate and build the mutants and run all SPEC experiments inside Docker containers on 3 *identical* machines, which are equipped with Intel® Core™ i7-4790 CPUs (3.60 GHz), 12 GiB RAM, and 256 GiB SSDs. The host runs Ubuntu 21.10 with Linux 5.13.0. Inside the Docker containers, we run an Ubuntu 20.04.3 LTS user-space. The experiments of `grep` are carried out on a commputer equipped with an AMD Ryzen Threadripper 2970WX 24-Core CPU, 64 GiB RAM, and a 1 TB NVMe SSD, which amplies the small performance effects of the P-mutants in case studies.

The workflow consists of the following steps: (1) we generate and store all P-mutants for all evaluation targets as source code, (2) we compile both the original programs (for the baseline) and the P-mutants using Clang with `O0` and `O3` optimization and store the resulting binaries, (3) we repeatedly execute both the original (baseline) and mutated binaries with their workloads and collect time measurements using GNU `time`. As shown in Table 4.2 and discussed in Section 4.4.2, SLOWCOACH generates hundreds of P-mutants for the evaluation targets. To reduce these numbers to a manageable level for experiment execution, we randomly sample 50 $\mathbb{Q}4$ P-mutants[6], and 30 random locations on which four $\mathbb{Q}3$ operators ($\mathbb{Q}3$-A to $\mathbb{Q}3$-D in Table 4.3) inject performance bugs. For each project in Table 4.2, there are a total of 170 P-mutants being sampled. Since performance measurements are affected by external factors [70], we repeat each execution 30 times and report median values if not stated otherwise. The median values are used because they are robust against outliers. To mitigate potential experiment stalls, we assign a time budget of 30 minutes for each execution, which is twice as long as the longest baseline execution needs.

4.4.2 RQ 1: Mutant Generation and Overheads

We analyze the amount of P-mutants SLOWCOACH generates for the evaluation targets when we apply four different $\mathbb{Q}3$ (effect simulation) and $\mathbb{Q}4$ (dev errors, no context) mutation operators (cf. Table 4.3). We omit $\mathbb{Q}2$ operators as they are reserved for the `grep` case study in Section 4.4.4. For $\mathbb{Q}3$ operators, Table 4.3 provides the *minimal performance overhead*

[6] If $\mathbb{Q}4$ operators produce less than 50 P-mutants, all generated P-mutants will be used.

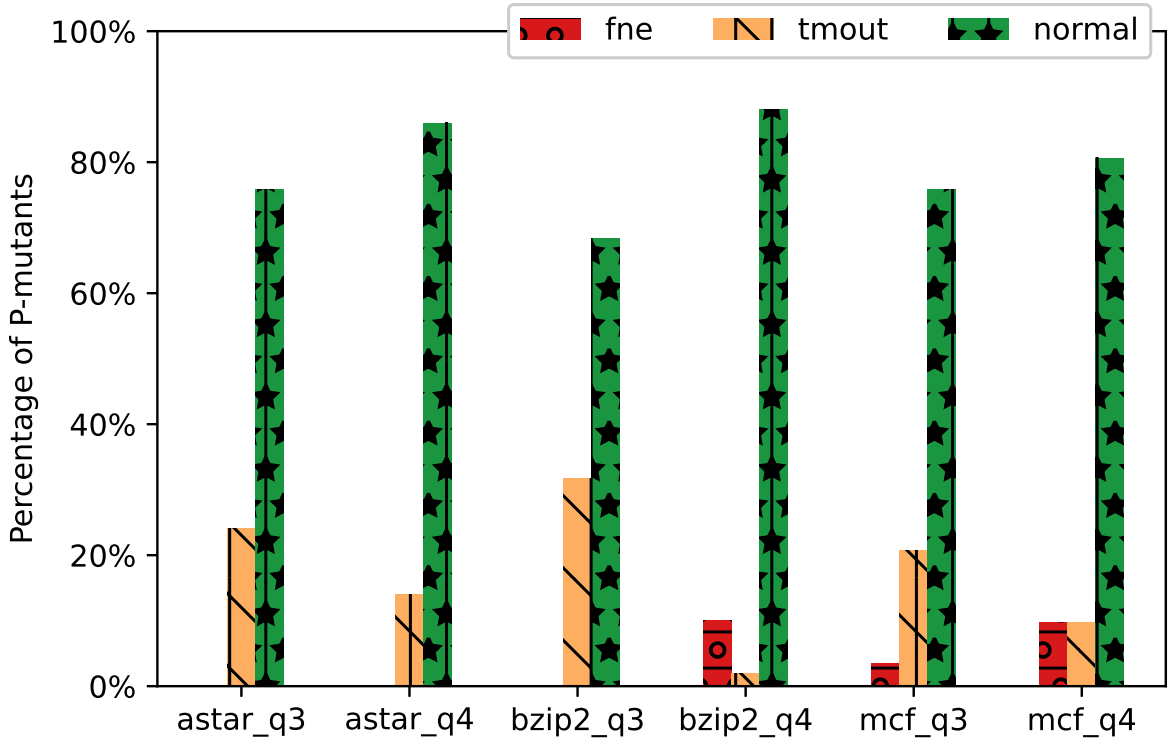


Fig. 4.4 Functional Equivalence by Operators and Programs

(MPO), which is the performance effect introduced in an individual execution of the mutated code, as measured with a microbenchmark on the mutated code chunks. The MPOs are given for both unoptimized (MPO_{00}) and optimized (MPO_{03}) compilation.

In total, SLOWCOACH produces 1645 ℚ3 and ℚ4 P-mutants for the three programs (`astar`, `bzip2` and `mcf`), as shown the *Mutants* column in Table 4.2. The T_m column shows the time SLOWCOACH takes to produce all source code P-mutants. The *OH* column is the total time SLOWCOACH takes to generate all source code files for P-mutants, inject sampled P-mutant source code files into the original project copies (cf. Section 4.4.1) and build sampled P-mutants. The time to produce P-mutants is measured with the debug version of SLOWCOACH without compiler optimization. SLOWCOACH produces source code P-mutants in less than a second and builds all sampled P-mutants within one minute. This shows the scalability of SLOWCOACH, where it generates 1645 P-mutants in 1.57 seconds and builds 491 P-mutants in 73.35 seconds. In summary, SLOWCOACH produces large amounts of P-mutants, where the exact number and distribution depends on the target program's code structure and the presence of performance optimization patterns in the code.

4.4.3 RQ 2: Functional Equivalence

In this section, we analyze the proportion of P-mutants which preserve FE compared to the original program as only those preserving FE are suitable for PMT. Since FE is undecidable for arbitrary programs, we resort to comparing the *standard output* (`stdout`) and the *standard error* (`stderr`) streams of baseline (original program) and P-mutant executions. If the outputs from both `stdout` and `stderr` of a P-mutant are identical to those of the baseline, this P-mutant is considered functionally equivalent. Otherwise the P-mutant is not considered to preserve FE and will not be used further as part of the mutation score computation in Section 4.4.4. Also, if a P-mutant terminates abnormally (exit with signals), this P-mutant is removed from further experiments. If a P-mutant does not finish within the assigned time budget, we consider it a timeout.

Figure 4.4 summarizes the results of my FE analysis based on O0 binaries. Among all 483 sampled P-mutants, we observe a total of 12 functional deviations (fne), 101 timeouts (tmout), and 370 P-mutants without functional deviation (normal). All 12 functional deviations are captured by signal 11 (segmentation fault), while none of them have caused output deviation. 4 instances of the segmentation faults are caused by the $\mathbb{Q}$3-C and $\mathbb{Q}$3-D operators that perform 1×10^5 and 1×10^6 iterations. They are all located at `pbeampp` in `mcf`, where large numbers of stack allocations (1×10^5 and 1×10^6 integers) lead to memory errors. Another 3 FE cases are caused by $\mathbb{Q}$4-A operators, and $\mathbb{Q}$4-B triggers the last 5 cases.

Timeouts can be seen as the *fuzzy part* between functional and performance bugs [8]. Timeouts can be caused either by a program hang, e.g., infinite loops from removing `break` statements (cf. Section 4.3.2), or by an excessively slow program. Without dedicated monitoring [39, 175], these cases are hard or impossible to distinguish.

In the experiments, about 20.9% of P-mutants yield timeouts, most notably by $\mathbb{Q}$3 operators. 94 out of 101 timeouts are fully optimized $\mathbb{Q}$3 P-mutants (`O3`), which do not timeout when unoptimized. From an example P-mutant in `bzip2`, we found that the unoptimized P-mutant finishes processing inputs in 144.65(12) s, while the fully optimized P-mutant binary takes 40 hours to finish. Further investigation with Linux `perf` shows that 27.15% of the CPU time of the unoptimized (`O0`) P-mutant is spent on the function where the redundant loop is injected, which is already the second slowest function according to `perf` reports. The fully optimized P-mutant, on the other hand, has spent 83.78% CPU time on the same function in the first hour of execution. By analyzing the assembly code of the affected `O3` binary, we see that the injected redundant loop is causing 76.91% of all CPU time. This finding demonstrates the potential of PMT techniques to be extended to identify compiler optimization anomalies because we observed optimized binaries to run longer than unoptimized

binaries. The mutation operators in SLOWCOACH could be applied as mutation approaches of compiler fuzzers [47], to facilitate the detection of compiler bugs or undefined behaviors.

We discard both P-mutants that functionally deviated or timed out for mutation score analysis, but keep the `O0` P-mutants whose `O3` counterparts timed out due to the anomaly where unoptimized binaries finish execution earlier than optimized binaries. In summary, SLOWCOACH is capable of generating 76.6% valid mutants for mutation score analysis.

4.4.4 RQ 3: Mutation Score and Discussion

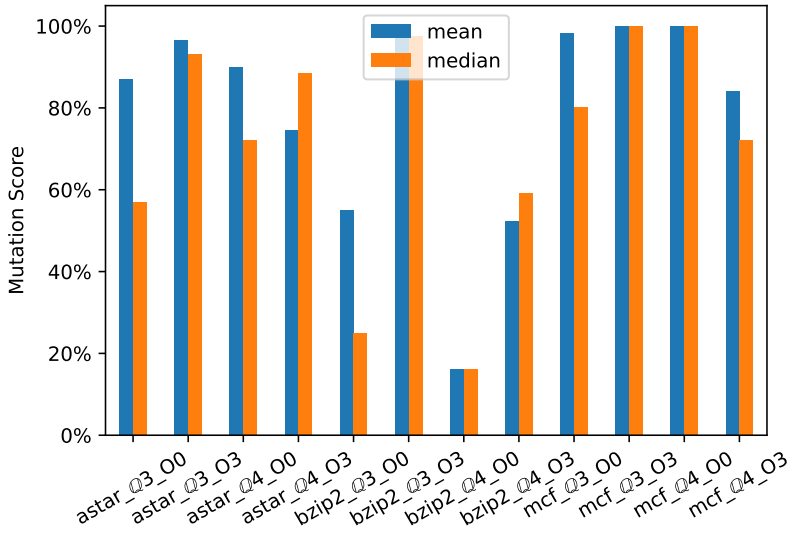
In this section, we evaluate SLOWCOACH by grading the performance testing setup (cf. Section 4.4.1) to help improve the quality of performance testing environments. The grade is defined to be the mutation score (as in classic MT) which is the number of P-mutants that can be killed. The wall clock time is used as metric to determine the performance of a mutant. As the wall clock time is susceptible to external noise, we repeat the execution 30 times and apply one-sided statistical testing to determine if a P-mutant can be killed.

There are many statistical approaches to compare the performance metric (wall clock time) results for a P-mutant P_m and for the baseline P_b, e.g., by arithmetic means or medians. For any workload whose $\overline{P_m} > \overline{P_b}$ or $median(P_m) > median(P_b)$, we can kill the P-mutant. But both means and medians are not robust enough to tolerate external noises. Besides using a single value to represent the performance, we can also use a two-sample KS-test [1] to kill P-mutants. Delgado-Pérez et al. [50] used Mann-Whitney U tests to compare P_m and P_b, by rejecting the null hypothesis that *P_m and P_b are drawn from the same distribution*. More specifically, the null hypothesis H_0 states that the cumulative distribution function (CDF) $F(x)$ of P_m is the same as the CDF $G(x)$ of P_b ($F(x) = G(x)$). If we can reject H_0 that P_m is drawn from the same distribution as the baseline P_b, a P-mutant is killed when its Mann-Whitney U test p-value is lower than the significance level α. We adopt 0.01, 0.05 and 0.1 as α values. As we are interested in results when the mutants perform slower than the baseline, this null-hypothesis could potentially be incorrectly rejected if we observe P_m which execute faster than P_b. Therefore, we adopt the one-sided KS-test [1] with the null hypothesis H_0', which states $F(x) \leq G(x)$. By rejecting H_0', we can assert that P_m is stochastically larger than P_b. If the P_m by any workload of a P-mutant is larger than P_b by the corresponding workload, then the P-mutant is killed. In summary, the previously discussed criteria are:

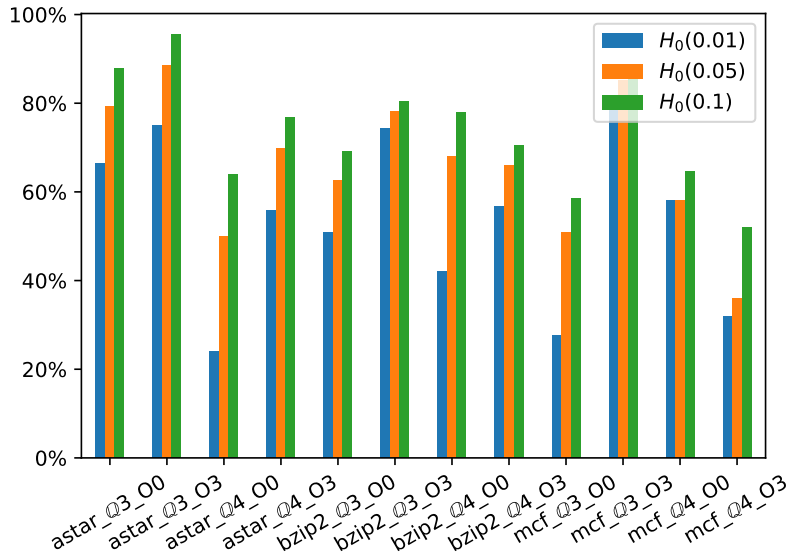
Cr1 $\overline{P_m} > \overline{P_b}$ to kill P-mutants

Cr2 $median(P_m) > median(P_b)$ to kill P-mutants

Cr3 H_0: $F = G$. Rejecting H_0 to kill P-mutants ($p < \alpha$).



(a) Mutation Scores by mean and median

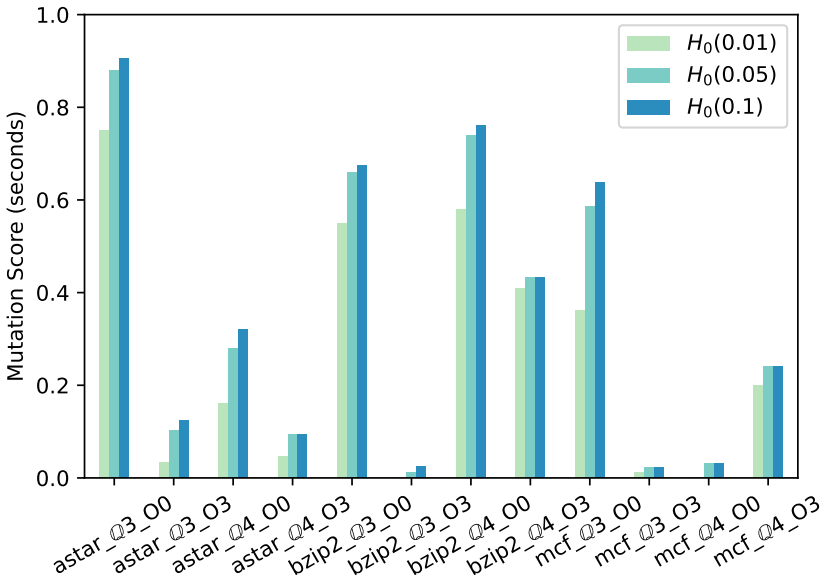


(b) Mutation Scores by H_0

Cr4 H_0': $F \leq G$. Rejecting H_0' to kill P-mutants ($p < \alpha$).

Figure 4.5 shows the mutation scores of the three programs by $\mathbb{Q}3$ and $\mathbb{Q}4$ operators and different optimization levels. The mutation score is computed as the percentage of killed P-mutants among all normal FE P-mutants without timeouts, defined as

$$score_{mut} = \frac{\text{number of killed P-mutants}}{\text{number of normal P-mutants}} . \tag{4.1}$$



(c) Mutation Scores by $H_0\prime$

Fig. 4.5 Mutation Scores of P-mutants by Operator Types and Programs

Figure 4.5a shows the mutation scores by arithmetic means (Cr1) and medians (Cr2), while Figures 4.5b and 4.5c show the mutation scores by H_0 (Cr3) and $H_0{}'$ (Cr4). The mutation score is measured in the interval of $[0,1]$, where 1 means all P-mutants are killed and 0 means no P-mutants are killed. We compute the overall mutations score for each criterion by applying the number of all killed P-mutants and that of all P-mutants across all 3 projects to Equation (4.1). The overall mutation scores by Cr1 to Cr4 are 0.81, 0.7, 0.76 and 0.42 respectively. Cr1 (mean) has the highest mutation score and Cr4 ($H_0{}'$) has the lowest, and the scores computed by Cr1 to Cr3 are relatively close to each other (about 0.75 ± 6). The overall mutation scores by Cr3 (Figure 4.5b) are 0.56, 0.69, and 0.76 with respect to significance levels 0.01, 0.05, and 0.1. These mutation scores are comparably close to those by Delgado-Pérez et al. [50] (0.51, 0.65, and 0.7).

The mutation scores by Cr4 (Figure 4.5c) are very different from those by Cr1 to Cr3 (Figures 4.5a and 4.5b). In Figure 4.5b, except for some cases, the mutation scores by O3 are mostly larger than those by O0. But for $H_0{}'$, the mutation score is the other way around except for mcf, where only 5% optimized P-mutants are killed while 37% unoptimized ones are killed. The largest difference in these experiments is shown by $\mathbb{Q}3$ P-mutants in `astar`, where unoptimized P-mutants have a score of 0.75 and optimized ones have 0.03 (the first two bars in Figure 4.5c). It is expected as the MPO values of unoptimized $\mathbb{Q}3$ operators are 10x slower than unoptimized ones (Table 4.3). Optimized P-mutants usually cannot expose the performance overheads which are quite obvious when unoptimized due to the small MPO.

The generic performance tool we use (`time`) to compute the mutation score suffers from too much noise and is not precise enough to measure millisecond performance impacts.

Among all unoptimized cases, 65% P-mutants are killed, while 11% optimized P-mutants are killed ($\alpha = 0.1$). $\mathbb{Q}3$ P-mutants, those in `astar` for example, show a large discrepancy of mutation scores by different optimization levels, where unoptimized P-mutants have a score of 0.9 and optimized ones have 0.05 (the first two bars in Figure 4.5c). It is expected as the MPO values of unoptimized $\mathbb{Q}3$ operators are 10 times slower than unoptimized ones (Table 4.3). $\mathbb{Q}4$ P-mutants are less likely to be killed and are more likely to be killed when unoptimized as well, except those by `mcf`.

We argue that PMT differs from classic MT in that PMT assesses the performance testing as a whole, rather than a particular workload or a profiler. Performance testing has many facets, including compiler optimizations, workload selection, selected performance diagnostic tools, or the repetition of the profiling, etc. The performance impacts by $\mathbb{Q}3$ and $\mathbb{Q}4$ P-mutants are believed to be small, unless repeated enough. The relatively higher mutation scores by unoptimized P-mutants (0.42) show that the P-mutants produced by SLOWCOACH do introduce performance overheads. Since software development is usually carried out with compiler optimization disabled, this also demontrates the capability of SLOWCOACH to assess the quality of the performance testing during the software's evolution.

The low mutation scores (overall 0.05%) by fully optimized P-mutants convey potential limitations of the performance testing enviornment. Firstly, the default workloads do not exercise the injected code frequently enough. This may be a scalability issue for production software as the release version of software is usually fully optimized, as the existing workloads cannot exhibit the performance bugs injected. Secondly, broader range of profilers with finer granularity need to be used in performance testing. The generic performance tool we use (`time`) to compute the mutation score suffers from too much noise and is not precise enough to measure millisecond performance impacts. As the MPO values in Table 4.3 illustrate, a single run of the injected code yields merely several milliseconds (sometimes even hundreds of microseconds) of performance overheads. Given external noises, small overheads are challenging to detect and profilers like `Linux perf` could be used to detect such overheads by sampling CPU [124].

Case Study on Context-dependent $\mathbb{Q}2$ Operators

SLOWCOACH also supports $\mathbb{Q}2$ operators to simulate real-world performance bugs given contextual information. We investigate the performance impact of such operators (cf. Section 4.3.2) in a case study on `grep`. They each produces a P-mutant which reproduces the scenarios involving performance bugs [27] and preserves FE. The performance impacts of the

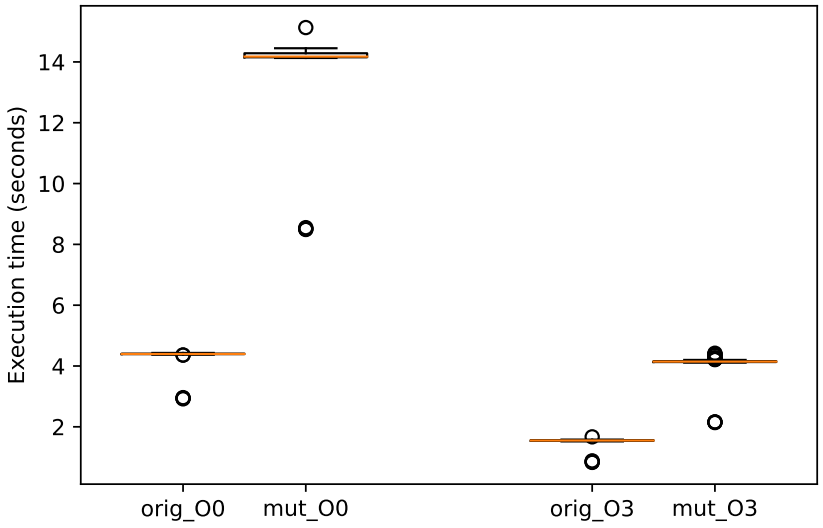
P-mutants are shown in Figure 4.6a. The y-axis is the execution wall clock time of the testing program reported in seconds. Experiments are repeated 50 times, and Figure 4.6b are the box plots of the performance values and their corresponding setup. Due to the small performance deviation, some boxes are overlapped with the median bar, and outliers are depicted as circles. Original program executions are labeled with *orig* and P-mutant executions with *mut*.

Since most real-world performance bugs involve performance bottlenecks for certain workloads, we use dedicated workloads for each mutant. In the first case ($\mathbb{Q}$2-A), we search a 50 MiB file with repeated "abcdabc" for the pattern `'abcd.bd'`. The performance impact of that mutant in the `O0` case is with 9.8 s median difference easy to detect. With `O3` optimization, the absolute impact is about 3.8 times smaller with 2.6 s, but still easily detectable. For the $\mathbb{Q}$2-B case, a file with 600 capitalized strings is matched with its uncapitalized strings in a multibyte locale. The developer who originally fixed the bug that $\mathbb{Q}$2-B re-introduces claims an overhead as large as 104 s. However, we observe median runtimes of only 0.41 s for `O0` and 0.23 s for `O3`. For the third case ($\mathbb{Q}$2-C), we apply the mutation to two different versions of `grep`: v3.4 (*mut* in Figure 4.6c) and v2.22 (*mut2*), and search a 10 MiB multibyte text file that contains encoding errors. According to developers, the mutant should yield a performance overhead of about 4.7 times. However, we can hardly observe a performance impact for v3.4, neither with `O0` nor `O3`. For the older v2.22, however, we observe a notable impact for `O0` with a relative overhead between medians of factor 3, which diminishes to factor 1.3 for `O3`.

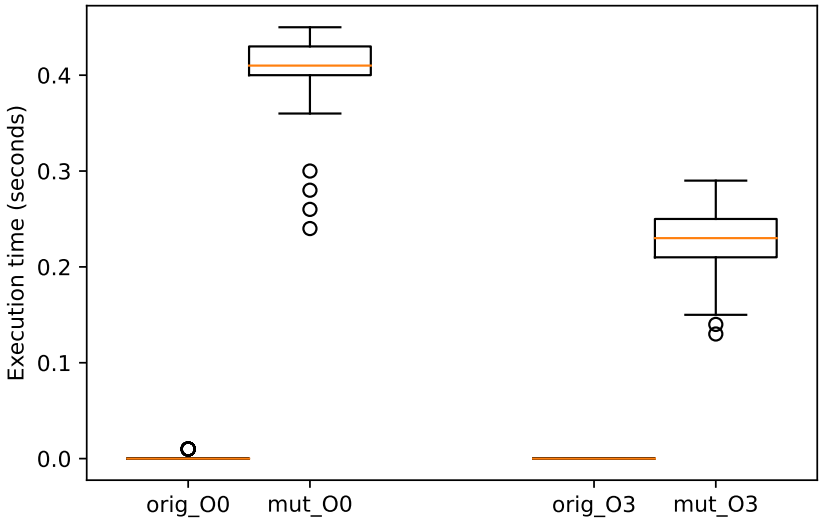
In summary, although $\mathbb{Q}$2 operators produce representative mutants derived from real-world bugs, their specificity leads to limited applicability and the required domain knowledge entails manual effort.

4.4.5 Internal and External Validity

The validity of the conclusions I draw in this work may be affected by several factors. The PMT operator selection could be biased as I focus my approach and evaluation on PMT operators derived from performance bugs that are widely discussed in the research community [156, 144], but there could be further classes of performance bugs that should be considered. The random sample of P-mutants and mutation score based on statistical testing could be statistically biased. My assessment of functional equivalence is based on program outputs, which assumes that the FE P-mutants yield identical outputs, which could be less robust. Bugs and mistakes in the implementation and data processing could also affect my conclusions. This is why I carefully tested and debugged the prototype implementation and all involved scripts and performed sanity checks on the collected data.



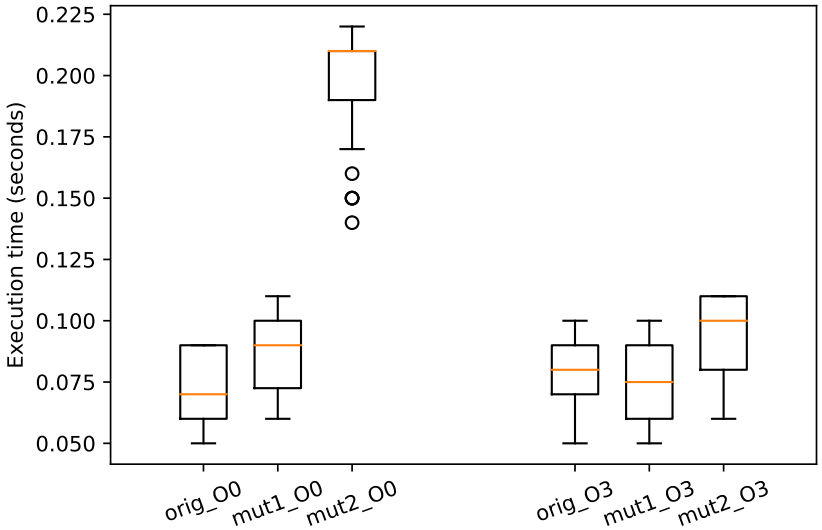
(a) Case Study ℚ2-A



(b) Case Study ℚ2-B

4.5 Conclusion

In this chapter, I presented and evaluated SLOWCOACH, a PMT framework. I subdivided the design space of PMT operators into four quadrants depending on whether they simulate effects of performance bugs or actual developer errors and identify functional equivalence as an key issue. I discuss concrete PMT operators from these quadrants and demonstrate how they can be derived from real-world performance optimizations and bugs. I demonstrate the applicability of my approach using 4 real-world software projects and show that the PMT operators can produce P-mutants that preserve functional equivalence and assess performance testing. I find that the mutation score based on one-sided statistical testing can provide reliable



(c) Case Study ℚ2-C

Fig. 4.6 Mutation Scores of P-mutants by Operator Types and Programs

assessments of the quality of the performance testing, which could help provide suggestions on improving performance testing.

Chapter 5

Performance Fuzzing

Feedback based fuzzing is a technique to automatically generate testing cases that cause unexpected software behaviors, e.g. program crashes. Recent researches aim to explore pathological inputs correlated with performance related issues, e.g., worst algorithmic cases that can be exploited to deploy denial of service (DoS) attacks. However, the performance fuzzing has fundamentally different characteristics in comparison with traditional fuzzing techniques and hence existing evaluation with default parameters by PERFFUZZ and SLOWFUZZ are insufficient. In this chapter, an empirical study on the efficacy of performance fuzzing is carried out, to identify conditions that improve the effectiveness of performance fuzzing. Besides, 2 PERFFUZZ variants are implemented to discuss the efficacy of performance fuzzing with different performance related pathological causalities. It turns out that the PERFFUZZ framework is limited due to the scope of the performance fuzzing guidance on internal factors of a program. Based on the performance impact analysis of generated inputs by 3 fuzzers on 4 popular projects, it shows that performance fuzzing should be executed with carefully selected parameters, preferably custom seed inputs and aligned compiler optimization levels. It is also found that the real timeout used by the fuzzer is usually lower than the one specified by users, and performance fuzzing would be limited by current internal information based guidance in terms of generating more interesting inputs. These conclusions are helpful to guide further development of more effective performance fuzzing tools in the future.

5.1 Introduction

Unexpected performance degradation of software leads to poor user experience, waste of computational resources and even Denial-of-Service attacks. For example, if a key-value data structure (like `dict` in python) is implemented as a bucket, and the hash algorithm is buggy

(XOR based), an attacker can deploy inputs with flipped bits to cause massive hash collisions, hence leaving the victim program hanging [46]. The common performance diagnostic procedure to fix such problems starts with performance bug detection [115, 84], causality localization [152, 124] and code optmization (c.f. Figure 1.4). Developers need to determine if there are performance bugs in the code as the first step, by one or more performance metrics of interests (PMoI), e.g., The wall clock time for a program to finish is too slow according to a specification. To simplify the discussion, the execution time or wall clock time is adopted as the PMoI throughout this chapter if not otherwise stated. Then developers run profilers to catpture details of the performance topology of the SUT for performance issue localization, for example, to show a prioritized list of functions by their percentage of CPU time [124]. The longer the CPU time a function takes, the more likely the causality code of the performance bottleneck dwells in this function. Developers can thereafter optimize the code around slow functions. Most state-of-the-art performance diagnose approaches proposed in research communities [148, 9, 163] only focus on the localization and profiling of predetermined performance issues. However, in the aforementioned key-value data structure example, developers may not be able to trivially identify the pathological input that triggers the worst algorithmic case. As performance bugs are known to be difficult to detect [84, 27] and reproduce [69], *finding proper inputs for performance testing* is one of the challenges to effectively find performance bugs.

In security testing, fuzzing is a popular technique adopted in recent years to generate pathological inputs[1] that crash the SUT [92, 126]. The fuzzing process starts with a set of initial *seeds inputs*, or *seeds* for short, and executes the SUT with each and every seeds while tracking the coverage of each input. Then the fuzzer randomly mutates the inputs and execute the SUT again. If the SUT crashes the fuzzer will record the causal input as an interesting input. Otherwise, the fuzzer will select those inputs that yield larger SUT code coverage, which is known as the *fuzzing guidance*, and repeat the inputs mutation and SUT execution until a preallocated time budget runs out. Performance fuzzing, PERFFUZZ in particular, extended the fuzzing guidance by counting the number of traversal over the SUT control flow graph (CFG) nodes [92], so that the fuzzer could explore the inputs that exploit *worst algorithmic cases*.

Both Lemieux et al. [92] and Petsios et al. [126] demonstrate that general performance fuzzing techniques are able to automatically generate effective inputs exploring very long execution paths. Nevertheless, both work failed to explain how generated inputs could be useful for software developers to identify and localize the performance issue causality in code.

[1]Though some researchers, e.g., Chen et al. [23], consider fuzzing as test case generation, test cases are considered as a SUT along with its inputs and many fuzzing tools like AFL fuzz around a fixated SUT.

Traditional fuzzing techniques search for vulnerabilities and record crashing inputs, which are usually small in numbers, and thus straightforward for developers to patch vulnerabilities by investigating each input. Performance fuzzing however, produces a set of inputs with increasing path length, and it remains unknown which one should devlopers investigate further for causality localization. A fuzzing process would usually produce thousands of inputs and it and it is unwise to manually numerate thousands of performance profiling traces. A naive approach to select an interesting input would be to investigate only the input with the highest fuzzing guidance value, e.g., the input with longest path length by PERFFUZZ. Despite the path length is correlated with the performance metrics like execution time or throughputs, CFG edges do not have the same execution cost. Figure 5.1 shows the execution time (y-axis) in seconds and the path length (x-axis)[2], where the slowest input yields 4 s, much more than the linear expectation (the green line) by the path length. As execution time or other PMoI are controlled by the fuzzing environment or configuration, previous researches [126, 92] have not discussed how fuzzing configuration could affect the fuzzer to generate *interesting inputs* for further performance analysis, which should a) be slow enough to be considered as a performance issue, and b) have *large performance impacts and small file sizes*.

As the measurement of PMoIs is susceptive to *external noises*, the first aim of this chapter is to *reevaluate* PERFFUZZ *to demonstrate how to apply performance fuzzing techniques effectively*. As Figure 5.1 suggests, the PMoI needed by performance bug localization may not be fully correlated to the approximation. The performance metrics that developers concern (e.g. execution time, throughputs, cache misses or I/O events etc), are measured externally to the SUT and hence recognized as *external factors*, but the fuzzing guidance is usually deployed to capture *internal factors* of the SUT, e.g. path length in PERFFUZZ [92] and SLOWFUZZ [126]. Lemieux et al. [92] also admitted that the execution path length may not be a good fuzzing guidance. Besides, more causalities such as worst algorithmic cases, bad cache coherence or inefficient I/O could lead to prolonged execution time as well. As PERFFUZZ [92] is designed to adopt various performance bottleneck causalities, the second aim of this study is *to explore the limitation of the existing performance fuzzing framework that prevents the performance fuzzer to identify more interesting inputs*.

In traditional fuzzing, fuzzers aim to iterate through inputs as fast as possible, so that fuzzers limit the file size and have a timeout to avoid getting stuck at a certain input. For example, the default parameters for `AFL` [169] are 1 MB for the input size and 1 s for the timeout. Such limits are deployed so that the fuzzing process could mutate and run as many inputs as possible given a time budget. On the other hand, the default limits on the

[2]Fuzzing and measuring the inputs by `libjpeg`.

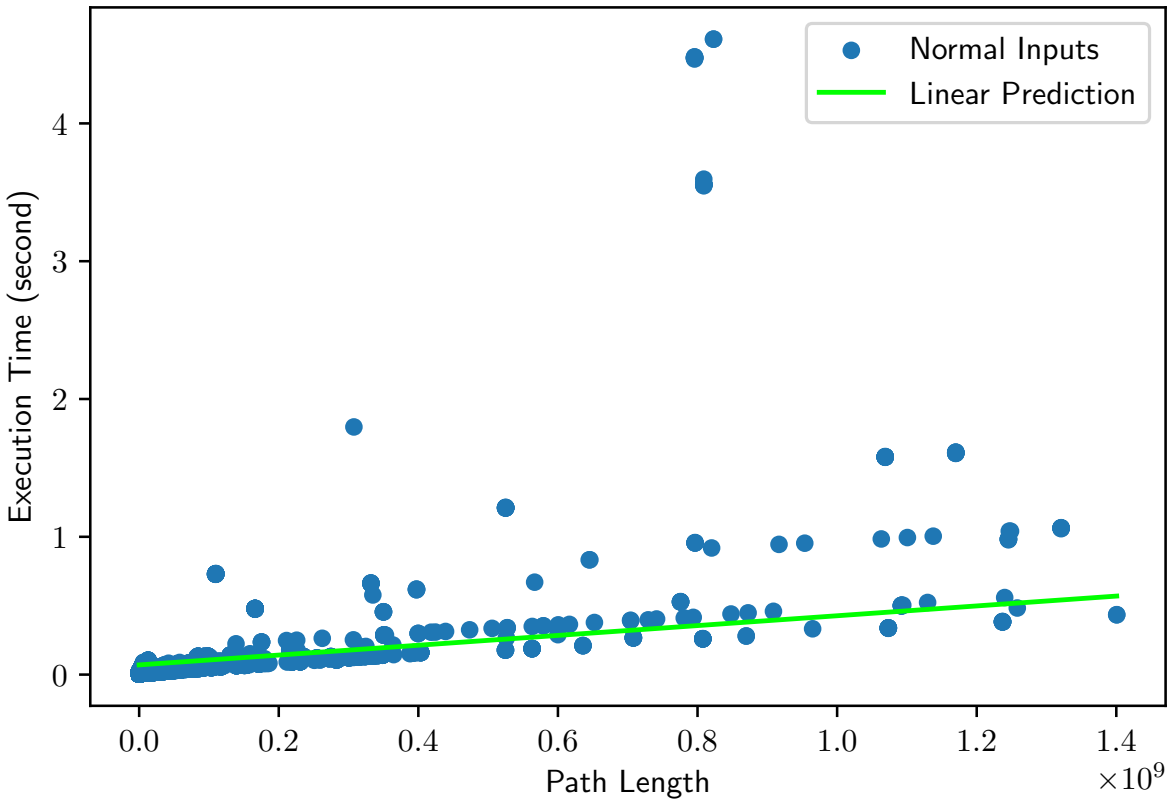


Fig. 5.1 Execution Time and Path Length

input size and timeout in performance fuzzing prevents the fuzzer to explore potentially interesting inputs. The default fuzzing parameters of PERFFUZZ are denoted as $\mathbb{S}1$, which has 1 s timeout and 1 MiB input size limit. The fuzzing timeout could lead to limited search space for performance fuzzing, as the execution time would increase with the path length in PERFFUZZ. The fuzzer would then consider a potentially interesting input as a hanging input, and stops searching further along its covered path.

To evaluate the impact of the fuzzing parameter, the timeout and the input size limit are increased by 100 times, to 100 s and 100 MiB, and this setup is denoted as $\mathbb{S}2$[3]. As it turns out in Section 5.4, that the fuzzing parameters by $\mathbb{S}1$ drastically limit the efficacy of performance fuzzing in finding slow inputs, the parameters by $\mathbb{S}2$ are adopted for the rest setups.

Furthermore, the selection of seed inputs usually affects the efficacy of fuzzing. To explore the impacts of the seed selection, several inputs are selected from the Internet for

[3]As AFL recommends a maximal input size value of 100 MiB, this value is considered as an input size bound for a larger search space to compare with. Since this value is 100 times of the default input size value, timeout is also scaled by 100 times.

each target project as customized seeds. The fuzzing setup with customized seeds are marked as $\mathbb{S}3$ and compared with $\mathbb{S}2$.

Another interesting scenario is the impact of the compiler optimization, which is never discussed in relevant communities. Build systems like `cmake` for example, will build the program without optimization at the debug mode, while build the program with full optimization at the release mode. Since compiler optimization would probably change the control flow structure, and hence the fuzzing guidance by CFG edges could be different on different unaligned optimization levels. Hence, the performance analysis on generated inputs could be impacted. In this chapter, an experiment setup in which the fuzzing is carried out on the unoptimized SUT and the performance measurement on the optimized SUT, is denoted as $\mathbb{S}4$ to be compared with $\mathbb{S}2$.

In addition to setups, this chapter also investigates two performance fuzzers with different fuzzing guidances, in order to identify the limits of the existing performance fuzzing framework. Both assign each CFG edge with a weight, given existing fuzzing guidances are based on CFG edges. One marks CFG edges with the number of instructions in its outbound basic block, while the other assigns the weight by the number of memory access instructions. These two variants are denoted as $\mathbb{V}1$ and $\mathbb{V}2$ respectively.

Summing up the previous discussion, this chapter aims to comparing different configurations of performance fuzzing, to provide suggestions on effective usage of performance fuzzing, and to advise challenges to be addressed in future researches on performance fuzzing. To simplify discussions, PERFFUZZ [92] is used as the target implementation, and execution time (as a common PMoI) is used to quantify the performance impact. More specifically, following reseach questions are discussed in this work:

RQ. 1 What is the impact of the default fuzzing parameters on performance fuzzing? ($\mathbb{S}1$ and $\mathbb{S}2$)

RQ. 2 What is the impact of seeds selection? ($\mathbb{S}3$ and $\mathbb{S}2$)

RQ. 3 What is the impact of compiler optimization? ($\mathbb{S}4$ and $\mathbb{S}2$)

RQ. 4 Could prioritizing the execution path (changing the fuzzing guidance) improve performance fuzzing? ($\mathbb{V}1$ and $\mathbb{V}2$ and $\mathbb{S}2$)

This chapter has the following contribution:

- A large scale evaluation on the PERFFUZZ framework.
- A set of suggestions to use and extend performance fuzzing effectively.

- Two PERFFUZZ variants exploring different aspects of performance bugs.

The key findings are:

- The timeout value would be the largest limit on the performance fuzzing.
- Custom seeds are considered relatively more effective than default ones regarding steering performance fuzzers to find more performance relevant inputs.
- The efficacy of performance fuzzing with unaligned optimization levels is relatively worse. So, it would always helpful to use a unified compiler optimization levels for performance fuzzing and performance localization/analysis.
- PERFFUZZ is limited in terms of guidance extension. Weighting execution paths yield relatively equivalently interesting inputs as default PERFFUZZ, but in some projects such weighted paths could misguide the fuzzer.

5.2 Background

The formal definition of the PERFFUZZ algorithm is detailed in Algorithm 1, which is a slightly simplified version by Lemieux et al. [92]. The general approach of performance fuzzing is to maximize the *performance value* (PV) associated with some *program components*. PERFFUZZ aims to collect the execution counts (as the PV) of control flow graph (CFG) edges (as the program component).

The PERFFUZZ algorithm starts by initializing $\mathscr{P}$ with the first set of inputs known as the *seed inputs* to (Line 3). Then the fuzzer will keep running until the time budget runs out (Line 3 - 11). During each iteration, the fuzzer computes the probability of each *input* from $\mathscr{P}$ (Line 4 and 5), and selects potentially *favored input*s (Line 6). Upon each selected input, PERFFUZZ mutates the it (Line 7). Mutants are thereafter streamlined to the SUT for execution and the fuzzer aquires the *feedback* (Line 8). If the feedback has new coverage or a new *maximal value* (Line 9), the mutants will be appended into the input set $\mathscr{P}$ (Line 10) for next iterations.

Line 9 shows the largest difference between the performance fuzzing and traditional fuzzing techniques. The code coverage of the generated inputs is more concerned by traditional grey-box fuzzing, while performance fuzzing also need to take the PV into account. In the case of PERFFUZZ, a mutant is *favored* if the cumulative sum of the execution count on CFG edges is larger than that by its direct parent input. And the probability if an input or a mutant is selected is defined in Equation (5.1) (Used at Line 5). In plain words, if

Algorithm 1 Fuzzing Algorithm [92]

```
1:  procedure PERFFUZZ
2:      𝒫 ← Seeds
3:      repeat
4:          for input in 𝒫 do
5:              p ← FUZZPROB(input)
6:              for i in SELECTINPUT(p, input) do
7:                  mutants ← MUTATE(i)
8:                  feedback ← EXECUTE(p, children)
9:                  if NEWCOV(feedback) ∨ NEWMAX(feedback) then
10:                     𝒫 ← 𝒫 ∪ mutants
11:     until timeout or signal received
```

an input has explored new paths or has found new PV (Line 9 in Algorithm 1), this input will be selected. Otherwise, there is a probability of α that the input will be selected, which is set to 0.01 by PERFFUZZ:

$$\text{FUZZPROB}(i) = \begin{cases} 1 & \text{if i is favored} \\ \alpha & \text{otherwise} \end{cases} \tag{5.1}$$

The procedure in Algorithm 1 and Equation (5.1) is briefly summarized in Figure 5.2. Performance fuzzing frameworks such as PERFFUZZ [92] can be extended with different fuzzing guidances (PVs) else than the path length. The flexibilty to look for new fuzzing guidances however, is limited by multiple factors. In addition, programming language features could also limit the extension of new fuzzing guidances. For example, memory allocation function invocations can be used to explore the inputs that trigger inefficient memory allocation, such as short-lived on-heap objects in C or C++. But many C or C++ projects customize project specific memory allocator else than `malloc()` in the standard library. Hence, a performance fuzzer needs contextual information of a project to guide the fuzzing process by some performance relevant factors like memory allocation or I/O events, which can hardly be tracked given merely the internal structure of a program.

In this chapter, two PERFFUZZ variants are proposed to explore the potential of PERFFUZZ as a framework. The PV, i.e., the path length used by PERFFUZZ, is an approximation of the real performance, because the classic fuzzing tools acquire the feedback information by instrumenting probes into the SUT (c.f. Section 5.1). Such approximation could deviate from the PMoI that developers are interested in, which are usually measured externally, e.g. execution time or throughputs etc. The deviation of path length from the real performance is caused by the fact that each CFG edge has different performance costs. A CFG edge could

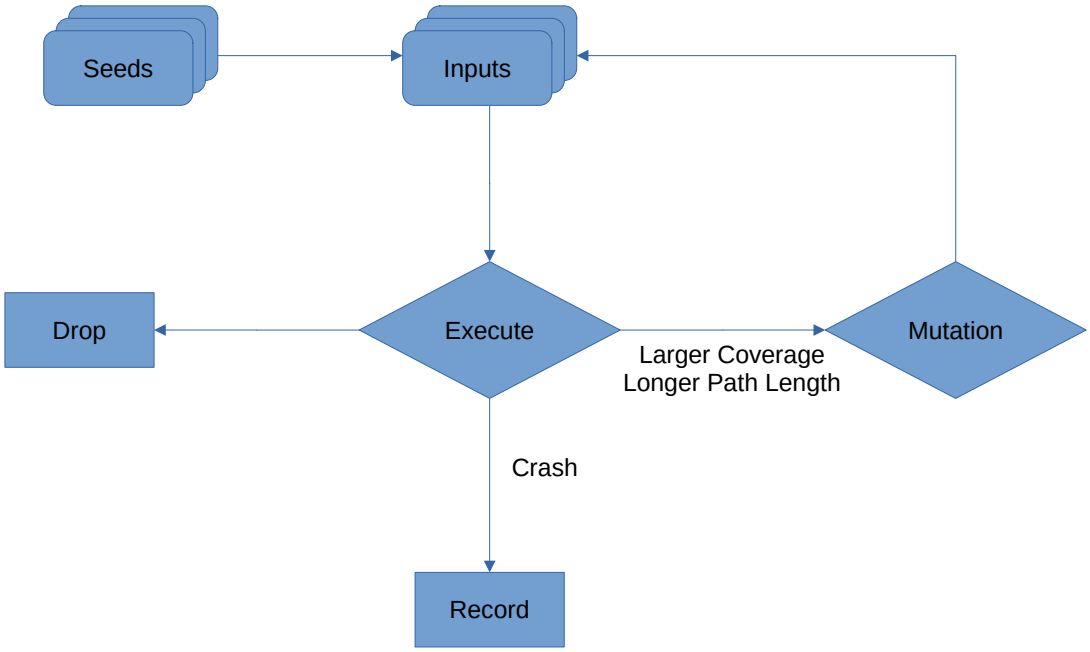


Fig. 5.2 A simplified common procedure of the fuzz testing

be traversed as quickly as in hundreds of cycles, and could be as slowly as in millions of cycles as well.

Two variants of PERFFUZZ are proposed to guide performance fuzzing concerning the performance cost of CFG edges. The two variant are denoted as $\mathbb{V}1$ and $\mathbb{V}2$ (c.f. Section 5.1). The hypothesis behind $\mathbb{V}1$ is that the performance of each CFG edge is correlated with the number of instructions execercised by its precedent basic block (CFG node). In plain words, the more instructions are executed, the slower the traversal of the CFG edge is. So, $\mathbb{V}1$ assigns each CFG edge with its number of instructions. Similarly, $\mathbb{V}2$ is based on the hypothesis that the accesses of memory play a key role in the performance of SUT. For example, summing a vector is usually faster than summing a list due to cache effects. When more memory accesses are triggered by an input, it is more likely that memory inefficiencies can be exposed. Hence, $\mathbb{V}2$ assigns each CFG edge with the number of underlying memory accesses. Both variants can be found at the github repository[4], where $\mathbb{V}1$ is the yapf-inst branch and $\mathbb{V}2$ is the yapf-mem-inst branch.

[4]https://github.com/61OlkVq8/perffuzz

Table 5.1 Evaluation Targets

Projects	SUT	LoC	seed size (default)	seed size ($\mathbb{S}3$)
libpng	readpng	94 750	1.22 KiB	35.17 KiB
zlib	minigzip -d	33 088	0.16 KiB	275.27 KiB
libjpeg	djpeg	66 521	0.40 KiB	381.11 KiB
libxml	xmllint	318 465	0.01 KiB	55.24 KiB

5.3 Study Design

PERFFUZZ is used for this study as a representative performance fuzzing framework. To compare the different configurations, The following evaluation targets are selected:

- `libpng-1.6.34`: reads a png file
- `zlib-1.2.11`: decompresses a file
- `libjpeg-turbo-1.5.3`: decompresses a jpeg file
- `libxml2-2.9.7`: parses an xml file

Table 5.1 provides the basic information of the target projects. These projects are selected thanks to their wide usage by many real world projects, and all evaluation target projects, experimental tasks as well as project versions are identical to those exercised by Lemieux et al. [92] for comparability. The execution time is chosen as the PMoI throughout this section and chapter[5].

The experimental process for each setup (c.f. Section 5.1) is described as: a) PERFFUZZ fuzzes the test program (SUT) of each project for 18 hours, with 8 instances running in parallel, and b) the SUT is executed with each generated input for 100 times and the execution time related to every input is recorded.

To compare different performance fuzzing configurations, the *Performance-Size Ratio (PSR)* of the slowest generated input and the percentage of *Performance Relevant Input (PRI)*s are discussed in this section, which are computed based on the execution time of generated inputs.

[5] In the rest of this chapter, PMoI can be considered as an equivalent of the execution time.

5.3.1 The Input with Worst Performance and Performance-Size Ratio (PSR)

A basic assumption to compare fuzzing configurations is that developers usually choose the input with the largest PMoI value (the slowest input) for further performance bottleneck identification and analysis. Hence, the slowest input is used as an indicator for the performance fuzzing efficacy:

$$PMoI_{max} = max(median(\bar{t_1}), median(\bar{t_1}) \dots median(\bar{t_n})) \textbf{ where } t_m \in T, m \in [1,n] \quad (5.2)$$

But, when the fuzzer incrementally mutate input payloads, the execution time increases linearly and a high execution time would be less interesting in this case. On contrast, if it takes a long time for the SUT to finish processing a small input, such input would demonstrate a certain case where the SUT could be optimized (or a DoS vulnerability to be exploited). Therefore, the gradient of the PMoI and the input size is considered as the indicator showing how interesting an underlying input is. Since PMoI is a set of values[6], the PMoI of an input is denoted as $\bar{t}$, and the set of PMoI of all inputs is denoted as T, where $\bar{t} \in T$. PSR is computed as the median of all PMoI values divided by the input size s:

$$PSR = \frac{median(\bar{t})}{size(t)} \quad (5.3)$$

Similarly, the PSR of this input is computed as in Equation (5.3) and denoted as PSR_{max}. If a setup has a higher PSR_{max} than another, this setup can be considered to be better in terms of finding a more interesting input.

5.3.2 Performance Relevant Input (PRI)

Another metric used to compare fuzzing configurations is to compute the percentage of PRI among all generated inputs. In fuzzing, all inputs except seeds are generated based on the mutation of their parental inputs. Given the performance of the ith input t_i, the performance of its selected descendants t_{t+1} should yield worse performance, or larger PMoI in other words. Such input is denoted as the performance relevant input (PRI). This dependency can be generalized as:

$$\forall P(\bar{t_i}) < P(\bar{t_{i+1}}) \textbf{ where } \{t_1, t_2, \dots, t_n\} \in T, i \in \{1, \dots, n-1\}, t_i \prec t_{i+1} \quad (5.4)$$

[6] By 100 repetitions, as mentioned earlier in this section.

T is the sequence of inputs $t_1, \ldots, t_n$, which are sorted by the order of the input creation, and $\bar{t_i}$ is the set of collected PMoI values. $P(t)$ is the cumulative probability function of the PMoI values, as the PMoI measurement is susceptive to external noises and usually PMoI values of an input would need repeated measures to avoid statistical bias. We adopt the two-sample K-S one-sided test to test if the set of PMoI denoted as $\bar{t_i}$ is stochastically less than t_{i+1}^{-} [1][7]. The null hypothesis H_0 is that $\bar{t_i} \geq t_{i+1}^{-}$. If H_0 is rejected, $\bar{t_i}$ is stochastically less than t_{i+1}^{-}. 0.01 is used as the confidence interval threshold α for the p value.

The percentage of PRIs indicates how effective the fuzzer searches for the inputs that increase PV, instead of looking for new coverage.

5.4 Evaluation

5.4.1 Evaluation Setup

The experiments are carried out on separate virtual machines with 8 Intel® Xeon® Gold 6248R CPUs (3.00 GHz) with 8 GiB memory, running Ubuntu Linux 20.04.3 LTS with the Linux kernel 5.4.0. All virtual machines are of the same configuration and running on SSDs with various spaces for each project to contain different sizes of generated inputs.

5.4.2 Overview

The results of aforementioned experiments are presented in Figure 5.3, Figure 5.4 and Figure 5.5. Figure 5.3 and Figure 5.4 show the $PMoI_{max}$ and PSR_{max} (computed by Equation (5.3)) of each project by every setup. The y-axis is the median of the execution time and *PSR* by the slowest input of a fuzzing process and is log scaled. The error bars demonstrate the results by repeated fuzzing process. The execution time median of all projects and setups can be found at Appendix B. Figure 5.5 shows the percentage of PRIs discussed in Section 5.3.2. The y-axis is the percentage of PRIs and is *linearly* scaled.

A general observation of the fuzzing results is that the fuzzing efficacy differ greatly among projects. The `libpng` project for example, can hardly execute for longer than 0.1 s, while the slowest input for `libxml` would need more than 30 s for the SUT to finish. Though `libpng` has 94k lines of code, its underlying SUT `readpng` is too trivial to have performance

[7] Delgado-Pérez et al. [50] use Mann-Whitney U Test [100] to compare stochastic performance data, which does not fit the performance comparison in this chapter as Mann-Whitney U test only checks the null hypothesis of the inequivalence of two samples. KS-test on the other hand is capable of rejecting the null hypothesis that one sample is stochastically larger or smaller than the other.

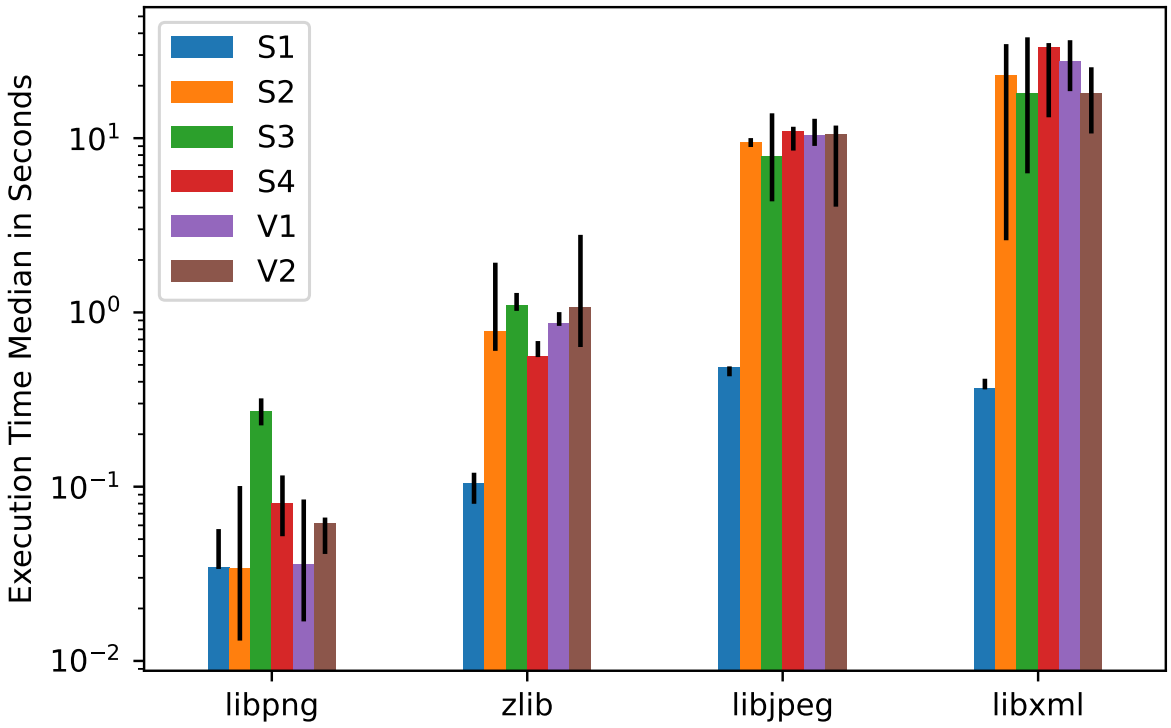


Fig. 5.3 Execution Time (Median) of the Slowest Input by Projects and Setups

impacts. The slowest input of `libpng` found by the fuzzer has a latency of merely 0.3 s by $\mathbb{S}3$.

5.4.3 Discussion

RQ. 1: What is the impact of the default fuzzing parameters on performance fuzzing? ($\mathbb{S}1$ and $\mathbb{S}2$)

Though the slowest input generated with default fuzzing parameters ($\mathbb{S}1$) yield comparable PSR values and PRI percentage with those by $\mathbb{S}2$, the absolute execution time is too small to be useful for performance analysis. For performance profilers such as `perf` [124], which aggregate the performance information by periodic sampling, such short execution time would not yield sufficient samples to exhibit the performance bottleneck location. In the case of `libjpeg`, $\mathbb{S}2$ shows the advantage of the larger search space, as PERFFUZZ found a slow input with a high PSR value.

In other projects, $\mathbb{S}2$ can find slow inputs but not potentially more interesting given PSR values and PRI percentages. This indicates that the performance impact of the slowest input found by $\mathbb{S}2$ is linearly correlated with the slowest input by $\mathbb{S}1$. It is likely that PERFFUZZ

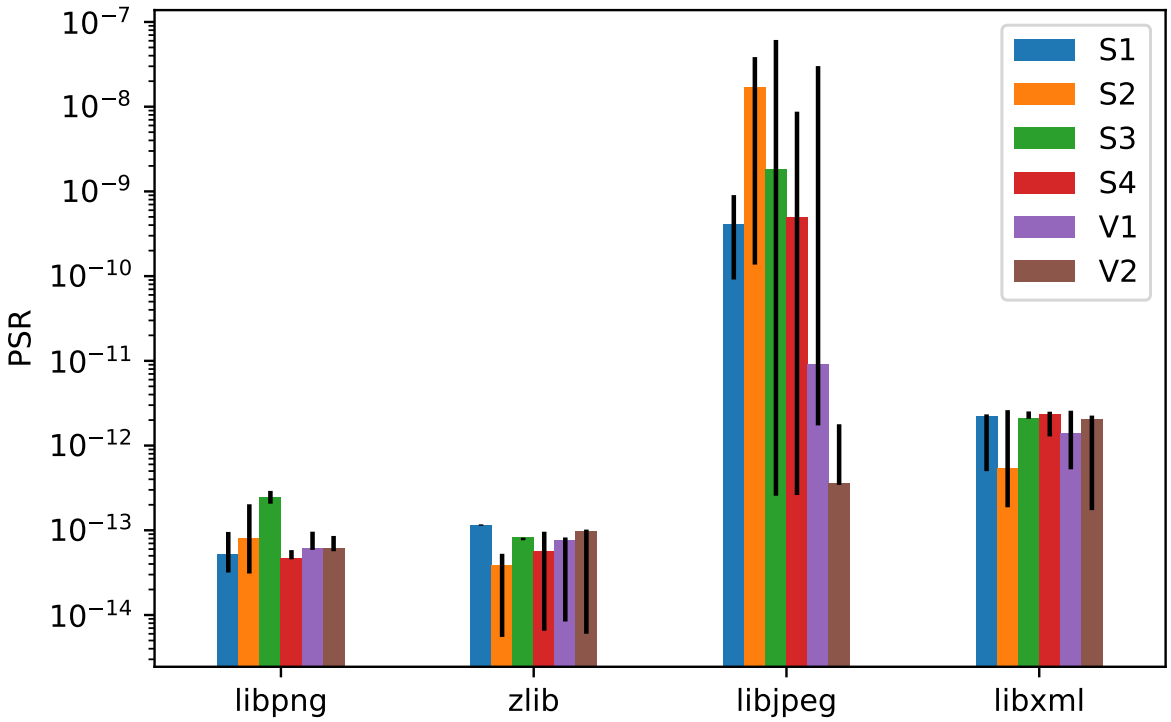


Fig. 5.4 PSR of the Slowest Input by Projects and Setups

would keep exploring linear paths until the input size or timeout limit is hit. As the main goal of performance fuzzing is to find the input that maximizes the performance impact while keeps its size reasonable, the input size limit by $\mathbb{S}2$ and other setups (100 MiB) is too large so that the fuzzer searched a lot of linear paths (c.f. Appendix B). To determine proper input size limits for different SUTs is intrinsically difficult to be generalized, as a too small limit would confine the fuzzing search space, while a too large limit would allow the performance fuzzer to explore less interesting linear cases.

The timeout value of the fuzzer provided by users (1 s for $\mathbb{S}1$ and 100 s otherwise) is *an upper bound* of real timeout values for every input, which are computed by the fuzzer dynamically [169]. Therefore, *user specified fuzzing parameters allow the fuzzer to explore larger space, but the timeout value cannot be fully utilized to search for slower inputs*.

RQ. 2: What is the impact of seeds selection? ($\mathbb{S}3$ **and** $\mathbb{S}2$)

Though custom seeds do not yield more interesting inputs in terms of $PMoI_{max}$ and PSR, custom seeds do have a very high percentage of PRIs. 60% to 80% of the inputs generated by $\mathbb{S}3$ are slower than their precedent inputs, while other setups yield merely 30% to 60% inputs to be PRIs. Hence, *using larger custom seeds is preferred*.

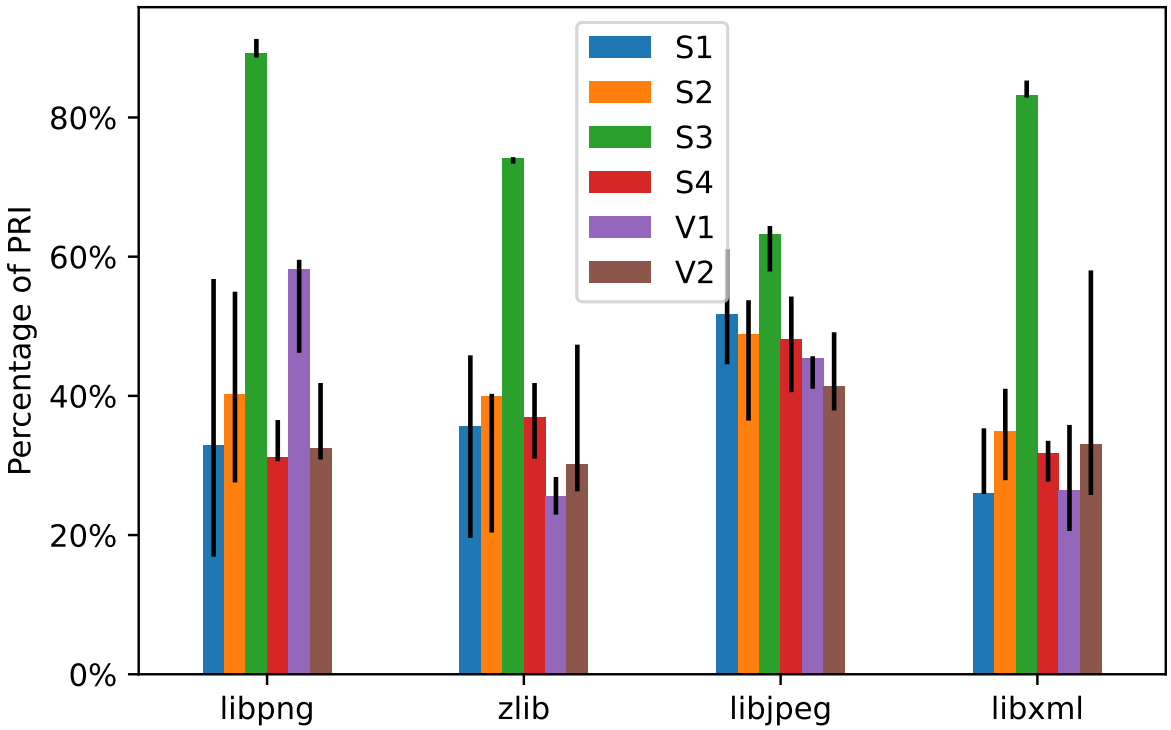


Fig. 5.5 PRI across Projects and Setups

RQ. 3: What is the impact of compiler optimization? ($\mathbb{S}4$ **and** $\mathbb{S}2$)

Though the compiler optimization changes CFG structures on which performance fuzzing is guided, the unaligned optimization levels between performance fuzzing and performance analysis do not affect the efficacy of performance fuzzing. The only exception is by `libjpeg` that the PSR value of the slowest input is moderately lower than that by $\mathbb{S}2$. Therefore, it is always recommended *to retain the optimization level applied by performance fuzzing when analyzing performance bottlenecks*.

RQ. 4: Could prioritizing the execution path (changing the fuzzing guidance) improve performance fuzzing? ($\mathbb{V}1$ **and** $\mathbb{V}2$ **and** $\mathbb{S}2$)

PERFFUZZ variants are able to generate inputs that are as slow as those by $\mathbb{S}2$, and the PSR values of these inputs are mostly close to the ones by $\mathbb{S}2$, except in the case of `libjpeg` where the PSR_{max} by $\mathbb{V}1$ and $\mathbb{V}2$ are lower than those by $\mathbb{S}2$. In other words, the proposed variants do not outperform the vanilla PERFFUZZ. The main reason is that, PERFFUZZ is guided by the number of traversals over CFG edges regardless how such edges are weighted. The variants like $\mathbb{V}1$ and $\mathbb{V}1$ may be able to find some CFG edges with performance impacts

earlier, but such CFG edges would also be found by PERFFUZZ given sufficient time. There are also technical concerns that $\mathbb{V}1$ and $\mathbb{V}2$ do not demonstrate more effectiveness over the vanilla PERFFUZZ. The technical limitation by $\mathbb{V}1$ is that the performance costs of LLVM instructions are different, just as the performance of CFG edges. The performance of memory access instructions by $\mathbb{V}2$ is volatile due to different cache effects. As the fuzzing guidance by PERFFUZZ is generalized to cover many cases, *computing the PV for fuzzing guidance based on CFG edges may not yield significant improvements to search for interesting inputs*.

5.4.4 Future Work

Summing up the previous discussions, two challenges for more effective performance fuzzing are identified:

1. The timeout is a major limit to be mitigated.
2. Performance fuzzing guidance by internal factors is limited and future research should target guiding performance fuzzing by external factors.

A natural solution woule be, to guide the fuzzing process by the execution time and increase the timeout value gradually, as suggested by Liščinský, Matúš [95]. However, the measurement of external factors like execution time is usually susceptive to noises. For example, the execution time would be drastically different if multiple instances of SUT are running in parallel, when CPU caches are saturated and contended. In the earlier experiments, the variance among repeated measures of the execution time can be 2 to 3 times larger than the normal exeuction time itself.

Since external noises cannot be completely nullified due to cache effects, OS scheduler etc, the fuzzing environment should be well controlled. More precisely, we cannot run multiple fuzzer instances in parallel. Moreover, because each measure of a performance metric would probably be different, it is unknown how many repetitions are needed to avoid statistic bias when determining NEWMAX at Line 9, Algorithm 1. Mytkowicz et al. [110] aim to determine the number of repetitions needed for performance comparison. But, a high repetition number would be too slow for a fuzzing process in terms of searching for new inputs. Hence, a balanced number of repetions should be considered while developing external factors guided performance fuzzing tools.

The aforementioned complication suggests significant changes of current performance fuzzing frameworks in the future work. Firstly, the new fuzzer needs the rework on timeout and hanging inputs decision logics so that the fuzzer could adaptively explore more potentially interesting inputs. Secondly, the new fuzzer needs a different fuzzer runtime. AFL [169]

based fuzzers inject the runtime code into the SUT to collect the coverage information, which is a large source of such noises for external factor measurements due to the massive memory accesses across the coverage map, and eventually affects the efficacy of performance fuzzing. The new fuzzer should avoid extra code instrumentation that collects internal states of the SUT, but rely more on performance counters by profilers such as `perf` [124] to guide the fuzzing[8].

5.5 Conclusion

In this chapter, an investigation on a performance fuzzing framework PERFFUZZ is carried out. Given different setups, performance fuzzing users are recommended to fine tune fuzzing parameters, use custom seed inputs and align the compiler optimization level for better fuzzing results. In addition, the limits of the PERFFUZZ framework are identified and suggestions to overcome these limits to develop better performance fuzzing tools are discussed.

[8]Despite most performance profilers also instrument runtime code into the SUT, such code is designed with minimal performance impacts [124, 152]

Chapter 6

Future Work

This thesis investigated performance bugs, PMT and performance fuzzing techniques. There are several related researches on performance bugs to reveil more insights into performance bug identification techniques as well as diagnostic tools.

As Chapter 3 has investigated over 700 performance bugs in the wild, a systematic reproduction of these bugs would support further researches. Firstly, the reproduction provides more corporus of real-world performance to be evaluated against, which provides more solid proof on the efficacy of the approaches under evaluation. Secondly, researchers could benefit from the performance bug reproduction to find more patterns, which help develop more sophisticated performance diagnostic tools, such as [144, 173]. A major challenge to this research would be the versatility of the scenarios that yield observable performance degradation. Like functional bug reproduction [86, 55, 13], performance bug reproduction suffers not only from limited information from bug reports, but also software and hardware impacts, and many performance bugs cannot be reproduced [69]

One of the major concern of the PMT framework in Chapter 4 is the FE checking. As previous work [78] already claimed, the FE for mutants is a problem that cannot be fully addressed. Hence, in Section 4.3.1 and Section 4.4.3, However, recent researches [65, 96] that involve more sophisticated equivalent mutant detection are worth of investigating.

As performance fuzzing tools show their efficacy in finding the algorithmic worst case, it would be an interesting topic to investigate how a fuzzing tool, such as PERFFUZZ, could find performance mutants generated by the PMT tool in Chapter 4. Since existing performance fuzzing tools, for example PERFFUZZ and SLOWFUZZ, are guided by the path length, the hypothesis is that these tools should be able to effectively exploit injected performance bugs by $\mathbb{Q}3$ P-mutants.

Performance fuzzing is a technique that can be further explored as suggested in Section 5.4.4. On one hand, the timeout value computed by the fuzzer should be revised to

adopt the increasing execution time of a newly generated input. On the other hand, it would be more effective to use external factors from performance profilers to guide the fuzzing. For example, we can use the hardware counter such as cache misses to guide the fuzzing that exploits the bad data organization of a program. Given the versatility of performance profilers like `perf`, more performance bug related symptoms can be identified and exploited by the future fuzzer.

Though this thesis focuses more on traditional logical approaches, future research should also be steered towards statistical based artificial intelligence (AI for short in the following text). Initial efforts to apply traditional machine learning algorithms [12] have explored performance bugs in the python language [157]. Deep learning algorithms [12, 59] demonstrate powerful functionality in improving fuzzing guidance [47]. The neural network, on which deep learning algorithms are based on, can also be used to train AI agents to behave smartly, which is known as reinforcement learning. Multi-agent reinforcement learning [5] combines the efforts of multiple AI agents, so that the AI system could find patterns that are previously ignored by human researchers. Performance bug researches could benefit from multi-agent reinforcement learning regarding performance bug patterns identification in the code.

Since early 2023, large language models (LLM) draw the attention from all over the world. LLMs based on transformers [165], such as ChatGPT [17, 158], is capable of talking to human beings as well as generating source codes upon human instructions. The powerful tool could also be helpful to investigate and identify performance bugs, even to help fix performance bugs.

Chapter 7

Conclusion

Software performance is an important attribute of software products. Performance bugs are software defects that degrade performance. Performance testing is able to pinpoint performance bugs in the code before the software is released and deployed. The performance testing procedure is composed of performance bug detection, localization, causality analysis and code optimization. Performance bug detection tells whether there is performance bug, while the performance bug localization involves the visualization of where the performance bottleneck dwells. Causality analysis and code optimization aim to reason the performance bug and to fix it. Performance bug detection and causality analysis are correlated and are of great significance towards the efficacy of performance testing.

There exists many tool supports for performance bug localization [61, 9, 152, 124]. However, these tools would only help developers when there is a performance bug being reported. A major challenge with performance bug detection is the lack of proper specification. This thesis aims to improve the quality of performance testing, by modeling and simulating performance specifications.

This thesis carried out an empirical study on real-world performance bugs with deep understanding of performance bugs semantics, in order to model a performance specification. In addition, a performance mutation testing (PMT) framework was developed in this thesis to grade a performance testing suite by injecting performance bugs into a program. An injected program is known as a *mutant*. If a performance testing suite is able to identify an injected bug, this mutant is considered as killed. The more mutants are killed, the more effective the underlying performance testing suites is expected. PMT hence helps developers to understand the capability of the performance testing suite in finding performance bugs.

Fuzzing is a popular approach that searches for program inputs, which could be used to generate test cases automatically. Recent researches also aim to apply the fuzzing technique on performance bugs [92]. However, different configurations could impact the efficacy of

performance fuzzing. This thesis investigated configurations as well as the potential extension of an existing performance fuzzing framework PERFFUZZ. The results show the limit of existing framework and provide suggestions on effective usage of performance fuzzing and future extension for more effective performance fuzzing.

Reseach Question 1: ***What are the characteristics of performance bugs?***

Performance testing tools are designed to detect performance bugs, caused by inefficient code. The first challenge of the performance tesing is the lack of testing oracles as in functional testing, which tell whether the underlying program runs correctly or incorrectly. To improve performance testing and model a reasonable performance specification, it is helpful to udnerstand the characteristics of performance bugs. Unfortunately, there is no dataset of known performance bugs available.

Contribution 1: ***A dataset and semantic taxonomy of known performance bugs.***

Chapter 3 presents a dataset of performance bugs in the real-world. The dataset classifies performance bugs into 8 categories by the semantics of bugs. This classification helps model performance bugs, and sees its usage in the performance mutation testing (PMT), detailed in Chapter 4. This dataset can be used as an assessment of the alignment of performance bug models and existing performance diagnostic tools. In addition, it can also provide a large body of instances for future research evaluation, thanks to its large number of performance bugs studied (over 700 bugs). Moreover, this dataset also studied characteristics of performance bugs in terms of performance bug complexity, which are used to prioritize certain types of performance bugs in the future research.

Research Question 2: ***How to determine if the performance testing is well calibrated to identify potential performance problems?***

In performance testing, testing setups could impact the performance bug detection. There is a need to determine whether factors including the selection of workload, the measurement of performance metrics are well established to detect possible performance bugs.

Contribution 2: ***A performance mutation testing (PMT) framework and a set of useful mutation operators.***

Mutation testing is used to test if a test suite is capable of finding functional bugs. Chapter 4 extends the idea of mutation testing to performance mutation testing. The mutation testing technique is based on *fault injection*, which aims to test the robustness of SUT. In mutation testing, if injected artificial errors can be detected by a test suite, this test suite should be robust enough to identify more complicated bugs.

Similarly, PMT injects performance bugs to verify the capability of a performance testing suite (benchmark and its workloads) to identify mismatch of the SUT performance and performance requirements. Chapter 4 also discusses performance bug fault models by context dependencies, on which mutation operators are implemented. The PMT framework in Chapter 4 adopts configurations to embed contextual information into mutation operators and is evaluated in 4 real-world projects.

Research Question 3: ***How to effectively use and adopt performance fuzzing?***

One of the conclusions by Chapter 4 is that the workload selection is important for the effectiveness of performance testing, as well as performance bug detection. So, performance fuzzing is investigated in Chapter 5, which is an automatic approach to properly generate inputs and test cases for performance analysis. The procedure of PERFFUZZ is a loop randomly mutating inputs and selecting inputs for the next iteration of random mutation, until the time budget runs out. In classic fuzzing, the inputs to be selected are those either yield larger code coverage, and any inputs crash the SUT are recorded. PERFFUZZ selects inputs by both code coverage and the number of traversals on CFG edges.

Although PERFFUZZ shows its capability in finding longer path length, it remains uncertain how generated inputs can be considered useful. In spite of reasonable approximation of the worst algorithmic case to the "bad" performance of a program, as performance bug localization and performance analysis relies on the performance metrics measured externally instead of the path length. Performance impacts can be measured from different aspects, such as execution time, throughput, etc. The *execution time* is used in this thesis, which is the most direct performance metrics related to the denial of service (DoS) attacks [46]. Since the efficacy of performance fuzzing is susceptible to many factors, it remains unclear how to use performance fuzzing effectively.

as admitted by Lemieux et al. [92], the path length may not be an ideal fuzzing guidance. It is therefore worth exploring, which fuzzing guidances could fit into existing performance frameworks (particularly PERFFUZZ), and what is the efficacy of these fuzzing guidances.

Contribution 3: ***An empirical study on the efficacy of* PERFFUZZ *with various fuzzing configurations and variants based on* PERFFUZZ**

PERFFUZZ [92] is used as the performance fuzzing target for the empirical study in Chapter 5. A total of 4 fuzzing configurations are compared, to investigate the effective usage of performance fuzzing. In addition, 2 PERFFUZZ based variants are implemented to explore if the performance fuzzing framework is flexible to adopt different performance bug symptoms.

The general recommendation to apply performance fuzzing is to customize fuzzing parameters, notably the timeout value as well as the file size limit. The result of this study also advises to use custom seed inputs and to align compiler optimization levels.

The result of PERFFUZZ variants shows that the PERFFUZZ framework is limited in terms of the fuzzing guidance, the PERFFUZZ is designed to be guided by internal factors of the SUT. The first problem is that the timeout value is dynamically computed and the user specified timeout is merely an upper bound. The second problem is that performance relevant information is *internal factors* that is not necessarily correlated with relevant external performance metrics. What's more, many performance related metrics require contextual information as those discussed in Chapter 4, such as the names of memory allocation functions to guide the fuzzing by memory allocation, as most C/C++ programs implement their own allocators for performance.

References

[1] *Kolmogorov–Smirnov Test*, pages 283–287. Springer New York, New York, NY, 2008. ISBN 978-0-387-32833-1. doi: 10.1007/978-0-387-32833-1_214.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811.

[3] Mejbah Alam, Justin Gottschlich, and Abdullah Muzahid. AutoPerf: A Generalized Zero-Positive Learning System to Detect Software Performance Anomalies. *arXiv:1709.07536 [cs]*, September 2017. URL http://arxiv.org/abs/1709.07536. arXiv: 1709.07536.

[4] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 298–313, New York, NY, USA, 2017. ACM. ISBN 9781450349383. doi: 10.1145/3064176.3064186.

[5] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2023. URL https://www.marl-book.com.

[6] Alejandro Cabrera Aldaya and Billy Bob Brumley. HyperDegrade: From GHz to MHz effective CPU frequencies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2801–2818, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/aldaya.

[7] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 422–435, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347716. doi: 10.1145/2991079.2991084. URL https://doi.org/10.1145/2991079.2991084.

[8] R. Atachiants, G. Doherty, and D. Gregg. Parallel performance problems on shared-memory multicore systems: Taxonomy and observation. *IEEE Transactions on Software Engineering*, 42(8):764–785, 2016. doi: 10.1109/TSE.2016.2519346.

[9] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, USA, 2012. USENIX Association. ISBN 9781931971966.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. doi: 10.1109/TDSC.2004.2.

[11] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. Xray: A function call tracing system. Technical report, 2016. A white paper on XRay, a function call tracing system developed at Google.

[12] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[13] Marcel Böhme and Abhik Roychoudhury. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 105–115, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2628058. URL http://doi.acm.org/10.1145/2610384.2628058.

[14] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134020. URL https://doi.org/10.1145/3133956.3134020.

[15] Daniel Pierre Bovet, Marco Cassetti, and Andy Oram. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., USA, 2000. ISBN 0596000022.

[16] Frederick P. Brooks. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201835959.

[17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[18] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982. doi: 10.1007/bf00625279. URL https://doi.org/10.1007/bf00625279.

[19] Marco Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014. doi: 10.1109/MSP.2014.66.

[20] Milind Chabbi, Shasha Wen, and Xu Liu. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 152–167, New York, NY, USA, 2018. ACM. ISBN 9781450349826. doi: 10.1145/3178487.3178499.

[21] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Mart: A mutant generation tool for llvm. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1080–1084, New York, NY, USA, 2019. ACM. ISBN 9781450355728. doi: 10.1145/3338906.3341180.

[22] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. Killing stubborn mutants with symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 30(2), January 2021. ISSN 1049-331X. doi: 10.1145/3425497.

[23] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.02.002. URL https://www.sciencedirect.com/science/article/pii/S0167404818300658.

[24] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243849. URL https://doi.org/10.1145/3243734.3243849.

[25] Jinfu Chen. *Performance Regression Detection in DevOps*, pages 206–209. ACM, New York, NY, USA, 2020. ISBN 9781450371223. doi: 10.1145/3377812.3381386.

[26] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1001–1012, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568259. URL http://doi.acm.org/10.1145/2568225.2568259. event-place: Hyderabad, India.

[27] Yiqun Chen, Stefan Winter, and Neeraj Suri. Inferring performance bug patterns from developer commits. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 70–81, 2019. doi: 10.1109/ISSRE.2019.00017.

[28] Yiqun Chen, Matthew Bradbury, and Neeraj Suri. Towards effective performance fuzzing. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 128–129, 2022. doi: 10.1109/ISSREW55968.2022.00055.

[29] Yiqun Chen, Oliver Schwahn, Roberto Natella, Matthew Bradbury, and Neeraj Suri. Slowcoach: Mutating code to simulate performance bugs. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 274–285, 2022. doi: 10.1109/ISSRE55969.2022.00035.

[30] Cisco Systems, Inc. Cisco Annual Internet Report (2018–2023) White Paper. Online: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, 2022. Accessed on 2022-12-28.

[31] Brendan Cody-Kenny, Michael O'Neill, and Stephen Barrett. Performance Localisation. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*, GI '18, pages 27–34, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5753-1. doi: 10.1145/3194810.3194815. URL http://doi.acm.org/10.1145/3194810.3194815.

[32] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, Berlin, Heidelberg, 1979. Springer-Verlag. ISBN 3540095101.

[33] Clang Community. libfuzzer – a library for coverage-guided fuzz testing. Online: https://www.llvm.org/docs/LibFuzzer.html, 2022. Accessed on 2022-11-25.

[34] The Linux Kernel Community. ftrace - function tracer — the linux kernel documentation. Online: https://www.kernel.org/doc/html/latest/trace/ftrace.html, . URL https://www.kernel.org/doc/html/latest/trace/ftrace.html.

[35] The Linux Kernel Community. Systemtap, . URL https://sourceware.org/systemtap.

[36] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. Memfuzz: Using memory accesses to guide fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58, 2019. doi: 10.1109/ICST.2019.00015.

[37] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN 0596005903.

[38] Domenico Cotroneo and Roberto Natella. Fault injection for software certification. *IEEE Security & Privacy*, 11(4):38–45, 2013. doi: 10.1109/MSP.2013.54.

[39] Domenico Cotroneo, Roberto Natella, and Stefano Russo. Assessment and improvement of hang detection in the linux operating system. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 288–294, 2009. doi: 10.1109/SRDS.2009.26.

[40] Domenico Cotroneo, Roberto Natella, and Roberto Pietrantuono. Predicting aging-related bugs using software complexity metrics. *Performance Evaluation*, 70(3): 163–178, 2013. ISSN 0166-5316. doi: https://doi.org/10.1016/j.peva.2012.09.004. URL https://www.sciencedirect.com/science/article/pii/S0166531612000946. Special Issue on Software Aging and Rejuvenation.

[41] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. A survey of software aging and rejuvenation studies. *J. Emerg. Technol. Comput. Syst.*, 10(1), jan 2014. ISSN 1550-4832. doi: 10.1145/2539117. URL https://doi.org/10.1145/2539117.

[42] Domenico Cotroneo, Anna Lanzaro, and Roberto Natella. Faultprog: Testing the accuracy of binary-level software fault injection. *IEEE Transactions on Dependable and Secure Computing*, 15(1):40–53, 2018. doi: 10.1109/TDSC.2016.2522968.

[43] Domenico Cotroneo, Antonio Ken Iannillo, Roberto Natella, and Roberto Pietrantuono. A comprehensive study on software aging across android versions and vendors. *Empirical Software Engineering*, 25(5):3357–3395, Sep 2020. ISSN 1573-7616. doi: 10.1007/s10664-020-09838-3. URL https://doi.org/10.1007/s10664-020-09838-3.

[44] Domenico Cotroneo, Antonio Ken Iannillo, Roberto Natella, and Stefano Rosiello. Dependability assessment of the android os through fault injection. *IEEE Transactions on Reliability*, 70(1):346–361, 2021. doi: 10.1109/TR.2019.2954384.

[45] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1476–1491, 2022. doi: 10.1109/TDSC.2020.3025289.

[46] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association. URL https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks.

[47] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 95–105, New York, NY, USA, 2018. ACM. ISBN 9781450356992. doi: 10.1145/3213846.3213848.

[48] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. The youtube video recommendation system. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, page 293–296, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589060. doi: 10.1145/1864708.1864770. URL https://doi.org/10.1145/1864708.1864770.

[49] Franciscone Luiz de Almeida, Renata Lopes Rosa, and Demostenes Zegarra Rodriguez. Voice quality assessment in communication services using deep learning. In *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, pages 1–6. IEEE, 2018.

[50] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. Performance mutation testing. *Software Testing, Verification and Reliability*, January 2020. doi: 10.1002/stvr.1728.

[51] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. *SIGPLAN Not.*, 50 (10):607–622, October 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814290.

[52] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, apr 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136. URL https://doi.org/10.1109/C-M.1978.218136.

[53] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Model-based mutant equivalence detection using automata language equivalence and simulations. *Journal of Systems and Software*, 141:1–15, 2018. ISSN 0164-1212. doi: 10.1016/j.jss.2018.03.010.

[54] Edsger W. Dijkstra. *Chapter I: Notes on Structured Programming*, page 1–82. Academic Press Ltd., GBR, 1972. ISBN 0122005503.

[55] Naji Dmeiri, David A Tomassi, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *Proceedings of the 41st International Conference on Software Engineering (ICSE) 2019 (to appear)*, 2019.

[56] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005. ISSN 1382-3256. doi: 10.1007/s10664-005-3861-2. URL http://dx.doi.org/10.1007/s10664-005-3861-2.

[57] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332132. doi: 10.1145/2663716.2663755. URL https://doi.org/10.1145/2663716.2663755.

[58] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201633612.

[59] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618.

[60] Google Inc. syzkaller - kernel fuzzer. Online: https://github.com/google/syzkaller, 2022. Accessed on 2022-11-24.

[61] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, jun 1982. ISSN 0362-1340. doi: 10.1145/872726.806987. URL https://doi.org/10.1145/872726.806987.

[62] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, USA, 1st edition, 2013. ISBN 0133390098.

[63] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 1st edition, 2019. ISBN 0136554822.

[64] Michael Grottke, Dong Seong Kim, Rajesh Mansharamani, Manoj Nambiar, Roberto Natella, and Kishor S. Trivedi. Recovery from software failures caused by mandelbugs. *IEEE Transactions on Reliability*, 65(1):70–87, 2016. doi: 10.1109/TR.2015.2452933.

[65] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 365–382, Cham, 2018. Springer International Publishing. ISBN 978-3-319-94144-8.

[66] Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 23:1–23:10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4427-2. doi: 10.1145/2961111.2962602. URL http://doi.acm.org/10.1145/2961111.2962602.

[67] Xue Han, Tingting Yu, and David Lo. Perflearner: Learning from bug reports to understand and generate performance test frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 17–28, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238204. URL https://doi.org/10.1145/3238147.3238204.

[68] Xue Han, Tingting Yu, and David Lo. PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 17–28, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3238204. URL http://doi.acm.org/10.1145/3238147.3238204. event-place: Montpellier, France.

[69] Xue Han, Daniel Carroll, and Tingting Yu. Reproducing performance bug reports in server applications: The researchers' experiences. *Journal of Systems and Software*, 156:268–282, October 2019. doi: 10.1016/j.jss.2019.06.100.

[70] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737.

[71] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a highly dependable operating system. In *Proceedings of the Sixth European Dependable Computing Conference*, EDCC '06, page 3–12, USA, 2006. IEEE Computer Society. ISBN 0769526489. doi: 10.1109/EDCC.2006.7. URL https://doi.org/10.1109/EDCC.2006.7.

[72] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, jul 2006. ISSN 0163-5980. doi: 10.1145/1151374.1151391. URL https://doi.org/10.1145/1151374.1151391.

[73] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, page 41–50, USA, 2007. IEEE Computer Society. ISBN 0769528554. doi: 10.1109/DSN.2007.46. URL https://doi.org/10.1109/DSN.2007.46.

[74] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 33–42, 2009. doi: 10.1109/DSN.2009.5270357.

[75] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In

Proceedings of the 16th International Conference on Software Engineering, ICSE '94, page 191–200, Washington, DC, USA, 1994. IEEE Computer Society Press. ISBN 081865855X.

[76] Intel Corporation. Intel® 64 and ia-32 architectures software developer manuals. Online: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2022. Accessed on 2022-11-24.

[77] Ray A Jarvis. A perspective on range finding techniques for computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2):122–139, 1983.

[78] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62.

[79] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 298–309, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213871. URL https://doi.org/10.1145/3213846.3213871.

[80] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 255–266, 2019. doi: 10.1109/ASE.2019.00033.

[81] Jiajun Jiang, Yingfei Xiong, and Xin Xia. A manual inspection of defects4j bugs and its implications for automatic program repair. *Science China Information Sciences*, 62 (10):200102, Sep 2019. ISSN 1869-1919. doi: 10.1007/s11432-018-1465-6. URL https://doi.org/10.1007/s11432-018-1465-6.

[82] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Fuzzing error handling code in device drivers based on software fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 128–138, 2019. doi: 10.1109/ISSRE.2019.00022.

[83] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using Context-Sensitive software fault injection. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2595–2612. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/jiang.

[84] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, 47(6):77–88, June 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254075.

[85] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and*

Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, pages 155–170, 2011. doi: 10.1145/2048066.2048081. URL https://doi.org/10.1145/2048066.2048081.

[86] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

[87] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882297. URL http://doi.acm.org/10.1145/1882291.1882297. event-place: Santa Fe, New Mexico, USA.

[88] Chung Hwan Kim, Junghwan Rhee, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Perfguard: Binary-centric application performance monitoring in production environments. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 595–606, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950347. URL http://doi.acm.org/10.1145/2950290.2950347.

[89] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. Pyse: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 136–147, 2019. doi: 10.1109/ICST.2019.00023.

[90] Xuan-Bach D. Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *SIGSOFT Softw. Eng. Notes*, 44(4):14, dec 2019. ISSN 0163-5948. doi: 10.1145/3364452.3364455. URL https://doi.org/10.1145/3364452.3364455.

[91] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming undefined behavior in llvm. *SIGPLAN Not.*, 52(6):633–647, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062343. URL https://doi.org/10.1145/3140587.3062343.

[92] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM. ISBN 9781450356992. doi: 10.1145/3213846.3213874.

[93] Wei Li, Jianping Yuan, Lu Zhang, Jie Cui, Xiaodong Wang, and Hua Li. semg-based technology for silent voice recognition. *Computers in Biology and Medicine*, 152: 106336, 2023. ISSN 0010-4825. doi: https://doi.org/10.1016/j.compbiomed.2022.106336. URL https://www.sciencedirect.com/science/article/pii/S0010482522010447.

[94] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568229. URL http://doi.acm.org/10.1145/2568225.2568229.

[95] Liščinský, Matúš. Fuzz testing of program performance [online], 2019. URL https://theses.cz/id/zw4694/.

[96] Nuno P. Lopes and José Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer*, 18(4):359–374, February 2015. doi: 10.1007/s10009-015-0366-1.

[97] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468.

[98] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 643–653, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635920. URL https://doi.org/10.1145/2635868.2635920.

[99] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Softw. Eng.*, 40(1): 23–42, jan 2014. ISSN 0098-5589. doi: 10.1109/TSE.2013.44. URL https://doi.org/10.1109/TSE.2013.44.

[100] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1): 50–60, March 1947. doi: 10.1214/aoms/1177730491. URL https://doi.org/10.1214/aoms/1177730491.

[101] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008. ISBN 0132350882.

[102] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., GBR, 2008. ISBN 0470343435.

[103] Jim Mauro and Richard McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, USA, 2006. ISBN 0131482092.

[104] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, USA, 2004. ISBN 0735619670.

[105] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR, USA, 2006. ISBN 0131568191.

[106] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a). *CoRR*, abs/1701.00854, 2017. URL http://arxiv.org/abs/1701.00854.

[107] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004. ISBN 0201702452.

[108] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL https://doi.org/10.1145/96267.96279.

[109] Elfurjani Sassi Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Software Testing*, 9, 1999. URL https://api.semanticscholar.org/CorpusID:15647411.

[110] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, mar 2009. ISSN 0362-1340. doi: 10.1145/1508284.1508275.

[111] Roberto Natella, Domenico Cotroneo, Joao A. Duraes, and Henrique S. Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2013. doi: 10.1109/TSE.2011.124.

[112] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3), February 2016. ISSN 0360-0300. doi: 10.1145/2841425.

[113] Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. Analyzing the effects of bugs on software interfaces. *IEEE Transactions on Software Engineering*, 46(3):280–301, 2020. doi: 10.1109/TSE.2018.2850755.

[114] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246, May 2013. doi: 10.1109/MSR.2013.6624035.

[115] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486862.

[116] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 902–912. IEEE Press, 2015. ISBN 9781479919345.

[117] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, jan 1992. ISSN 1049-331X. doi: 10.1145/125489.125473. URL https://doi.org/10.1145/125489.125473.

[118] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 369–378, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737966. URL http://doi.acm.org/10.1145/2737924.2737966. event-place: Portland, OR, USA.

[119] Oracle. Oracle Linux DTrace Reference Guide. Online: https://docs.oracle.com/en/operating-systems/oracle-linux/dtrace-guide, 2022. Accessed on 2022-11-24.

[120] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330576. URL https://doi.org/10.1145/3293882.3330576.

[121] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360600. URL https://doi.org/10.1145/3360600.

[122] Aditya Pakki and Kangjie Lu. Exaggerated error handling hurts! an in-depth study and context-aware detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1203–1218, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417256. URL https://doi.org/10.1145/3372297.3417256.

[123] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 2018.

[124] Perf Maintainers. Perf Wiki. Online: https://perf.wiki.kernel.org, 2022. Accessed on 2022-04-22.

[125] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. *Does Mutation Testing Improve Testing Practices?*, pages 910–921. IEEE Press, 2021. ISBN 9781450390859. doi: 10.1109/ICSE43902.2021.00087.

[126] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, New York, NY, USA, 2017. ACM. ISBN 9781450349468. doi: 10.1145/3133956.3134073.

[127] J. Pradeep, K. Vijayakumar, and M. Harikrishnan. *2 Voice Recognition Using Natural Language Processing*, pages 15–24. 2021.

[128] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 554–569, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359640. URL https://doi.org/10.1145/3341301.3359640.

[129] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[130] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer, 2011.

[131] Marc J. Rochkind. *Advanced UNIX Programming (2nd Edition)*. Addison Wesley Longman Publishing Co., Inc., USA, 2004. ISBN 0131411543.

[132] Nadav Rotem, Lee Howes, and David Goldblatt. Warrior1: A performance sanitizer for c++, 2020.

[133] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Search-based Mutation Testing to Improve Performance Tests. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, pages 316–317, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5764-7. doi: 10.1145/3205651.3205670. URL http://doi.acm.org/10.1145/3205651.3205670.

[134] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Tandem: A taxonomy and a dataset of real-world performance bugs. *IEEE Access*, 8: 107214–107228, 2020. doi: 10.1109/ACCESS.2020.3000928.

[135] Charitha Saumya, Jinkyu Koo, Milind Kulkarni, and Saurabh Bagchi. Xstressor : Automatic generation of large-scale worst-case test inputs by inferring path conditions. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 1–12, 2019. doi: 10.1109/ICST.2019.00011.

[136] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[137] Ali Sedaghatbaf, Mahshid Helali Moghadam, and Mehrdad Saadatmand. Automated performance testing based on active deep learning. *CoRR*, abs/2104.02102, 2021. URL https://arxiv.org/abs/2104.02102.

[138] John A. Sharp. Data oriented program design. *SIGPLAN Not.*, 15(9):44–57, sep 1980. ISSN 0362-1340. doi: 10.1145/947706.947713. URL https://doi.org/10.1145/947706.947713.

[139] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 270–281, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. doi: 10.1145/2771783.2771816. URL https://doi.org/10.1145/2771783.2771816.

[140] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. Industry practice of coverage-guided enterprise linux kernel fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 986–995, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3340460. URL https://doi.org/10.1145/3338906.3340460.

[141] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-123-6. doi: 10.1145/1082983.1083147. URL http://doi.acm.org/10.1145/1082983.1083147.

[142] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136, New York, NY, USA, 2000. ACM. ISBN 1-58113-195-X. doi: 10.1145/350391.350420. URL http://doi.acm.org/10.1145/350391.350420.

[143] Linhai Song and Shan Lu. Statistical Debugging for Real-world Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 561–578, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660234. URL http://doi.acm.org/10.1145/2660193.2660234. event-place: Portland, Oregon, USA.

[144] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 370–380, 2017. doi: 10.1109/ICSE.2017.41.

[145] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 982–993. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00103.

[146] M. Sujon, M. Shafiuzzaman, M. M. Rahman, and R. Rahman. Characterization and localization of performance-bugs using Naive Bayes approach. In *2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 791–796, May 2016. doi: 10.1109/ICIEV.2016.7760110.

[147] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Nature, 2022.

[148] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. *SIGPLAN Not.*, 45(5):269–280, January 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693489.

[149] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, USA, 3rd edition, 2007. ISBN 9780136006633.

[150] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., USA, 2005. ISBN 0131429388.

[151] The GNU Project. Savannah Git Hosting - grep.git. Online: https://git.savannah.gnu.org/cgit/grep.git, 2022. Accessed on 2022-04-20.

[152] The LTTng Project. LTTng: an open source tracing framework for Linux. Online: https://lttng.org, 2022. Accessed on 2022-04-22.

[153] Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 189–199, New York, NY, USA, 2020. ACM. ISBN 9781450380089. doi: 10.1145/3395363.3404540.

[154] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 314–326, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356176. doi: 10.1145/3168830. URL https://doi.org/10.1145/3168830.

[155] Catia Trubiani, Alexander Bran, André van Hoorn, Alberto Avritzer, and Holger Knoche. Exploiting load testing and profiling for Performance Antipattern Detection. *Information and Software Technology*, 95:329–345, March 2018. ISSN 0950-5849. doi: 10.1016/j.infsof.2017.11.016. URL http://www.sciencedirect.com/science/article/pii/S0950584917302276.

[156] Sokratis Tsakiltsidis, Andriy Miranskyy, and Elie Mazzawi. On automatic detection of performance bugs. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, oct 2016. doi: 10.1109/issrew.2016.43. URL https://doi.org/10.1109%2Fissrew.2016.43.

[157] Sokratis Tsakiltsidis, Andriy V. Miranskyy, and Elie Mazzawi. Towards automated performance bug identification in python. *CoRR*, abs/1607.08506, 2016. URL http://arxiv.org/abs/1607.08506.

[158] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[159] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng.

[160] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017. doi: 10.1109/SP.2017.23.

[161] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316699. doi: 10.1145/2349896.2349905. URL https://doi.org/10.1145/2349896.2349905.

[162] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 260–275, New York, NY, USA, 2013. Association for Computing

Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522728. URL https://doi.org/10.1145/2517349.2522728.

[163] Shasha Wen, Xu Liu, and Milind Chabbi. Runtime value numbering: A profiling technique to pinpoint redundant computations. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 254–265, 2015. doi: 10.1109/PACT.2015.29.

[164] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. Watching for software inefficiencies with witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 332–347, New York, NY, USA, 2018. ACM. ISBN 9781450349116. doi: 10.1145/3173162.3177159.

[165] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6.

[166] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.32. URL https://doi.org/10.1109/FOSE.2007.32.

[167] Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv.*, 44(3), jun 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187679. URL https://doi.org/10.1145/2187671.2187679.

[168] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 389–400, New York, NY, USA, 2016. ACM. ISBN 9781450343909. doi: 10.1145/2931037.2931070.

[169] Michal Zalewski. American fuzzy lop (2.52b). Online: https://lcamtuf.coredump.cx/afl/, 2022. Accessed on 2022-04-22.

[170] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security Versus Performance Bugs: A Case Study on Firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985457. URL http://doi.acm.org/10.1145/1985441.1985457.

[171] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software*

Repositories, MSR '12, pages 199–208, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1761-0. URL http://dl.acm.org/citation.cfm?id=2664446.2664477.

[172] Shujie Zhao, Yiqun Chen, Stefan Winter, and Neeraj Suri. Analyzing and improving customer-side cloud security certifiability. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 300–307, 2019. doi: 10.1109/ISSREW.2019.00088.

[173] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. Wperf: Generic off-cpu analysis to identify bottleneck waiting events. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 527–543, USA, 2018. USENIX Association. ISBN 9781931971478.

[174] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. A systematic literature review of how mutation testing supports quality assurance processes. *Software Testing, Verification and Reliability*, 28(6):e1675, 2018. doi: https://doi.org/10.1002/stvr.1675. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1675. e1675 stvr.1675.

[175] Yian Zhu, Yue Li, Jingling Xue, Tian Tan, Jialong Shi, Yang Shen, and Chunyan Ma. What is system hang and how to handle it. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 141–150, 2012. doi: 10.1109/ISSRE.2012.12.

[176] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/zou.

Appendix A

Case Studies in grep

This section provides more reference details of the examples inspiring the $\mathbb{Q}2$ mutation operators in Section 4.3.2.

A.1 Case 1

Full ID `3255bc58e8fb2d98145dbb2dd17bae0a5e47a85e`

Reproduce `yes abcdabc | head -50000000 >k; env LC_ALL=C time -p src/grep -i 'abcd.bd' k`

Code Change

```
diff --git a/src/dfa.c b/src/dfa.c
index 8e6c708..fc638c8 100644
--- a/src/dfa.c
+++ b/src/dfa.c
@@ -3411,6 +3411,28 @@ dfahint (struct dfa *d, char const *begin, char *end, size_t *count)
     }
  }

+bool
+dfaisfast (struct dfa *d)
+{
+  size_t i;
+  if (d->superset)
+    {
+      for (i = 0; i < d->superset->tindex; i++)
+        if (d->superset->tokens[i] == BACKREF)
+          return false;
+      return true;
+    }
+  else if (!d->multibyte)
+    {
```

```
+      for (i = 0; i < d->tindex; i++)
+        if (d->tokens[i] == BACKREF)
+          return false;
+      return true;
+    }
+  else
+    return false;
+}
+
 static void
 free_mbdata (struct dfa *d)
 {
diff --git a/src/dfa.h b/src/dfa.h
index 60aff11..2dd4871 100644
--- a/src/dfa.h
+++ b/src/dfa.h
@@ -82,6 +82,10 @@ extern char *dfaexec (struct dfa *d, char const *begin, char *end,
 extern size_t dfahint (struct dfa *d, char const *begin, char *end,
                        size_t *count);

+/* Return true, if `multibyte' attribute of struct dfa is false and the
+   pattern doesn't have BACKREF.  */
+extern bool dfaisfast (struct dfa *);
+
 /* Free the storage held by the components of a struct dfa. */
 extern void dfafree (struct dfa *);

diff --git a/src/dfasearch.c b/src/dfasearch.c
index dc76397..79a0cd8 100644
--- a/src/dfasearch.c
+++ b/src/dfasearch.c
@@ -213,6 +213,7 @@ EGexecute (char const *buf, size_t size, size_t *match_size,
   size_t len, best_len;
   struct kwsmatch kwsm;
   size_t i;
+  bool dfafast = dfaisfast (dfa);

   mb_start = buf;
   buflim = buf + size;
@@ -232,13 +233,13 @@ EGexecute (char const *buf, size_t size, size_t *match_size,
               beg += offset;
               /* Narrow down to the line containing the candidate, and
                  run it through DFA. */
-              end = memchr (beg, eol, buflim - beg);
-              end = end ? end + 1 : buflim;
               match = beg;
               beg = memrchr (buf, eol, beg - buf);
               beg = beg ? beg + 1 : buf;
               if (kwsm.index < kwset_exact_matches)
                 {
+                  end = memchr (beg, eol, buflim - beg);
+                  end = end ? end + 1 : buflim;
                   if (MB_CUR_MAX == 1)
                     goto success;
                   if (mb_start < beg)
```

```
@@ -253,17 +254,21 @@ EGexecute (char const *buf, size_t size, size_t *match_size,
                                    &backref))
                       continue;
                   }
-               else
+               else if (!dfafast)
                   {
+                    /* Narrow down to the line we've found if dfa isn't fast. */
+                    end = memchr (match, eol, buflim - beg);
+                    end = end ? end + 1 : buflim;
                     if (dfahint (dfa, beg, (char *) end, NULL) == (size_t) -1)
                       continue;
                     if (! dfaexec (dfa, beg, (char *) end, 0, NULL, &backref))
                       continue;
                   }
              }
-           else
+           if (!kwset || dfafast)
              {
-               /* No good fixed strings; start with DFA. */
+               /* No good fixed strings or DFA is fast; start with DFA
+                  broadly. */
                size_t offset, count;
                char const *next_beg;
                count = 0;
```

A.2 Case 2

Full ID `960ad317db21e781b04010f4128bb149273a3327`

Reproduce `time -p env LC_ALL=C src/grep -Fivf pat in; time -p env LC_ALL=ja_JP.eucjp src/`

Code Change

```
diff --git a/src/dfasearch.c b/src/dfasearch.c
index c2e0177..548ef08 100644
--- a/src/dfasearch.c
+++ b/src/dfasearch.c
@@ -89,7 +89,7 @@ kwsmusts (void)
   struct dfamust *dm = dfamust (dfa);
   if (!dm)
     return;
-  kwsinit (&kwset);
+  kwsinit (&kwset, false);
   if (dm->exact)
     {
       /* Prepare a substring whose presence implies a match.
diff --git a/src/grep.c b/src/grep.c
index fc22c7b..ae3b6e7 100644
--- a/src/grep.c
+++ b/src/grep.c
@@ -2265,14 +2265,91 @@ contains_encoding_error (char const *pat, size_t patlen)
   return false;
 }

+/* The set of wchar_t values C such that there's a useful locale
+   somewhere where C != towupper (C) && C != towlower (towupper (C)).
+   For example, 0x00B5 (U+00B5 MICRO SIGN) is in this table, because
+   towupper (0x00B5) == 0x039C (U+039C GREEK CAPITAL LETTER MU), and
+   towlower (0x039C) == 0x03BC (U+03BC GREEK SMALL LETTER MU).  */
+static short const lonesome_lower[] =
+  {
+    0x00B5, 0x0131, 0x017F, 0x01C5, 0x01C8, 0x01CB, 0x01F2, 0x0345,
+    0x03C2, 0x03D0, 0x03D1, 0x03D5, 0x03D6, 0x03F0, 0x03F1,
+
+    /* U+03F2 GREEK LUNATE SIGMA SYMBOL lacks a specific uppercase
+       counterpart in locales predating Unicode 4.0.0 (April 2003).  */
+    0x03F2,
+
+    0x03F5, 0x1E9B, 0x1FBE
+  };
+
+static bool
+fgrep_icase_available (char const *pat, size_t patlen)
+{
+  for (size_t i = 0; i < patlen; ++i)
+    {
+      unsigned char c = pat[i];
+      if (localeinfo.sbclen[c] > 1)
```

```
+        return false;
+    }
+
+  for (size_t i = 0; i < patlen; ++i)
+    {
+      unsigned char c = pat[i];
+
+      wint_t wc = localeinfo.sbctowc[c];
+      if (wc == WEOF)
+        return false;
+
+      wint_t uwc = towupper (wc);
+      if (uwc != wc)
+        {
+          char s[MB_LEN_MAX];
+          mbstate_t mb_state = { 0 };
+          size_t len = wcrtomb (s, uwc, &mb_state);
+          if (len > 1 && len != (size_t) -1)
+            return false;
+        }
+
+      wint_t lwc = towlower (uwc);
+      if (lwc != uwc && lwc != wc && towupper (lwc) == uwc)
+        {
+          char s[MB_LEN_MAX];
+          mbstate_t mb_state = { 0 };
+          size_t len = wcrtomb (s, lwc, &mb_state);
+          if (len > 1 && len != (size_t) -1)
+            return false;
+        }
+
+      for (size_t j = 0; lonesome_lower[j]; j++)
+        {
+          wint_t li = lonesome_lower[j];
+          if (li != lwc && li != uwc && li != wc && towupper (li) == uwc)
+            {
+              char s[MB_LEN_MAX];
+              mbstate_t mb_state = { 0 };
+              size_t len = wcrtomb (s, li, &mb_state);
+              if (len > 1 && len != (size_t) -1)
+                return false;
+            }
+        }
+    }
+
+  return true;
+}
+
 /* Change a pattern for fgrep into grep.  */
 static void
-fgrep_to_grep_pattern (size_t len, char const *keys,
-                       size_t *new_len, char **new_keys)
+fgrep_to_grep_pattern (char **keys_p, size_t *len_p)
 {
-  char *p = *new_keys = xnmalloc (len + 1, 2);
```

```
+  char *keys, *new_keys, *p;
   mbstate_t mb_state = { 0 };
-  size_t n;
+  size_t len, n;
+
+  len = *len_p;
+  keys = *keys_p;
+
+  new_keys = xnmalloc (len + 1, 2);
+  p = new_keys;

   for (; len; keys += n, len -= n)
     {
@@ -2300,7 +2377,13 @@ fgrep_to_grep_pattern (size_t len, char const *keys,
         }
     }

-  *new_len = p - *new_keys;
+  free (*keys_p);
+  *keys_p = new_keys;
+  *len_p = p - new_keys;
+
+  matcher = "grep";
+  compile = Gcompile;
+  execute = EGexecute;
 }

 int
@@ -2733,20 +2816,17 @@ main (int argc, char **argv)
       In a multibyte locale, switch from fgrep to grep if either
       (1) case is ignored (where grep is typically faster), or
       (2) the pattern has an encoding error (where fgrep might not work).  */
-  if (compile == Fcompile
-      && (MB_CUR_MAX <= 1
-          ? match_words
-          : match_icase || contains_encoding_error (keys, keycc)))
+  if (compile == Fcompile)
     {
-      size_t new_keycc;
-      char *new_keys;
-      fgrep_to_grep_pattern (keycc, keys, &new_keycc, &new_keys);
-      free (keys);
-      keys = new_keys;
-      keycc = new_keycc;
-      matcher = "grep";
-      compile = Gcompile;
-      execute = EGexecute;
+      if (MB_CUR_MAX > 1)
+        {
+          if (contains_encoding_error (keys, keycc))
+            fgrep_to_grep_pattern (&keys, &keycc);
+          else if (match_icase && !fgrep_icase_available (keys, keycc))
+            fgrep_to_grep_pattern (&keys, &keycc);
+        }
+      else if (match_words)
```

```
+         fgrep_to_grep_pattern (&keys, &keycc);
      }

    compile (keys, keycc);
diff --git a/src/kwsearch.c b/src/kwsearch.c
index 508ebc5..7fe08aa 100644
--- a/src/kwsearch.c
+++ b/src/kwsearch.c
@@ -38,7 +38,7 @@ Fcompile (char const *pattern, size_t size)
 {
   size_t total = size;

-  kwsinit (&kwset);
+  kwsinit (&kwset, true);

   char const *p = pattern;
   do
diff --git a/src/search.h b/src/search.h
index 431a67d..534a49e 100644
--- a/src/search.h
+++ b/src/search.h
@@ -47,7 +47,7 @@ _GL_INLINE_HEADER_BEGIN
 typedef signed char mb_len_map_t;

 /* searchutils.c */
-extern void kwsinit (kwset_t *);
+extern void kwsinit (kwset_t *, bool);
 extern ptrdiff_t mb_goback (char const **, char const *, char const *);
 extern wint_t mb_prev_wc (char const *, char const *, char const *);
 extern wint_t mb_next_wc (char const *, char const *);
diff --git a/src/searchutils.c b/src/searchutils.c
index 8081d41..87f51a4 100644
--- a/src/searchutils.c
+++ b/src/searchutils.c
@@ -25,15 +25,33 @@
 #define NCHAR (UCHAR_MAX + 1)

 void
-kwsinit (kwset_t *kwset)
+kwsinit (kwset_t *kwset, bool mb_trans)
 {
   static char trans[NCHAR];
   int i;

-  if (match_icase && MB_CUR_MAX == 1)
+  if (match_icase && (MB_CUR_MAX == 1 || mb_trans))
     {
-      for (i = 0; i < NCHAR; ++i)
-        trans[i] = toupper (i);
+      if (MB_CUR_MAX == 1)
+        for (i = 0; i < NCHAR; ++i)
+          trans[i] = toupper (i);
+      else
+        for (i = 0; i < NCHAR; ++i)
+          {
```

```
211 +             wint_t wc = localeinfo.sbctowc[i];
212 +             wint_t uwc = towupper (wc);
213 +             if (uwc != wc)
214 +               {
215 +                 char s[MB_LEN_MAX];
216 +                 mbstate_t mbs = { 0 };
217 +                 size_t len = wcrtomb (s, uwc, &mbs);
218 +                 if (len > 1)
219 +                   abort ();
220 +                 trans[i] = s[0];
221 +               }
222 +             else
223 +               trans[i] = i;
224 +           }
225
226           *kwset = kwsalloc (trans, false);
227       }
```

A.3 Case 3

Full ID `5cb49d2f375f0606ac9d916af6024d4b92ba0786`

Reproduce `yes \$(printf '\200\200\200\200\200\200x\n')`

head -n 1000000 >j; grep -oP y j|

Code Change

```
diff --git a/src/pcresearch.c b/src/pcresearch.c
index 8f3d935..c0b8678 100644
--- a/src/pcresearch.c
+++ b/src/pcresearch.c
@@ -229,6 +229,7 @@ Pexecute (char *buf, size_t size, size_t *match_size,
            while (mbclen_cache[to_uchar (*p)] == (size_t) -1)
              {
                p++;
+               subject = p;
                bol = false;
              }

@@ -269,29 +270,30 @@ Pexecute (char *buf, size_t size, size_t *match_size,
              }
            int valid_bytes = sub[0];

-           /* Try to match the string before the encoding error.  */
-           if (valid_bytes < search_offset)
-             e = PCRE_ERROR_NOMATCH;
-           else if (valid_bytes == 0)
+           if (search_offset <= valid_bytes)
              {
-               /* Handle the empty-match case specially, for speed.
-                  This optimization is valid if VALID_BYTES is zero,
-                  which means SEARCH_OFFSET is also zero.  */
-               sub[1] = 0;
-               e = empty_match[bol];
              }
-           else
-             e = jit_exec (subject, valid_bytes, search_offset,
-                           options | PCRE_NO_UTF8_CHECK | PCRE_NOTEOL, sub);
+               /* Try to match the string before the encoding error.  */
+               if (valid_bytes == 0)
+                 {
+                   /* Handle the empty-match case specially, for speed.
+                      This optimization is valid if VALID_BYTES is zero,
+                      which means SEARCH_OFFSET is also zero.  */
+                   sub[1] = 0;
+                   e = empty_match[bol];
+                 }
+               else
+                 e = jit_exec (subject, valid_bytes, search_offset,
```

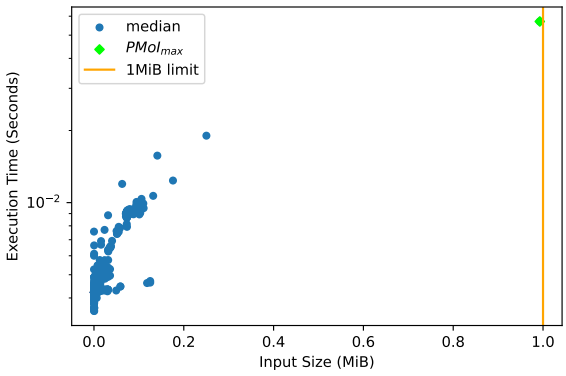
```
+                                options | PCRE_NO_UTF8_CHECK | PCRE_NOTEOL, sub);

-          if (e != PCRE_ERROR_NOMATCH)
-            break;
+            if (e != PCRE_ERROR_NOMATCH)
+              break;
+
+            /* Treat the encoding error as data that cannot match.  */
+            p = subject + valid_bytes + 1;
+            bol = false;
+          }

-          /* Treat the encoding error as data that cannot match.  */
           subject += valid_bytes + 1;
-          if (p < subject)
-            p = subject;
-          bol = false;
         }

       if (e != PCRE_ERROR_NOMATCH)
```

Appendix B

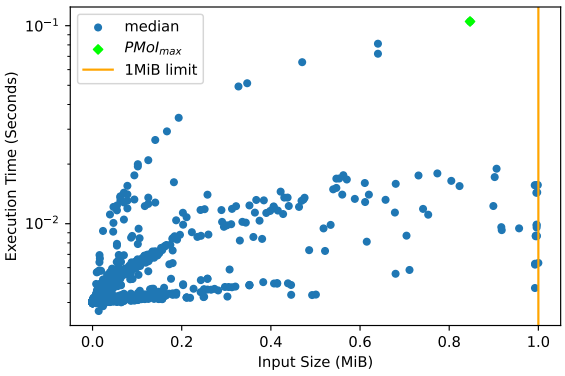
Complete Performance Fuzzing Experiment Data

This appendix contains all execution time medians of all generated inputs by target projects, setups and repetitions.

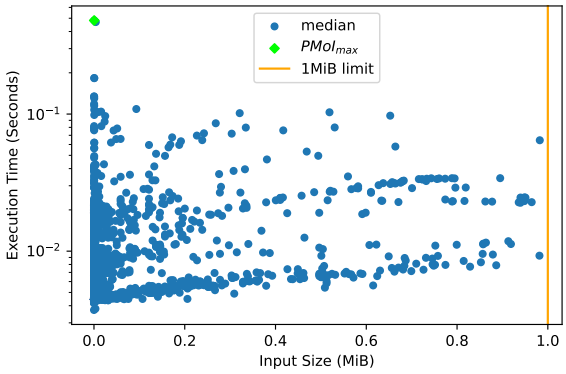
B.1 $\mathbb{S}1$



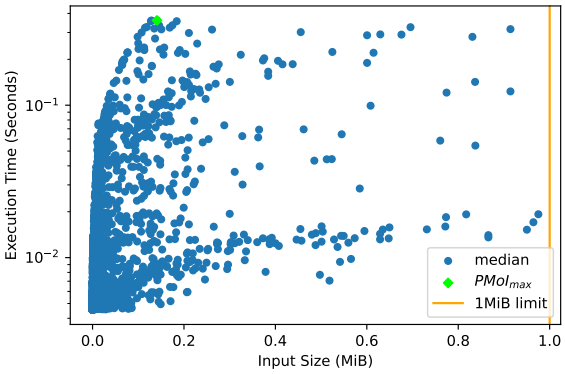
(a) `libpng` $\mathbb{S}1$, Rep 1



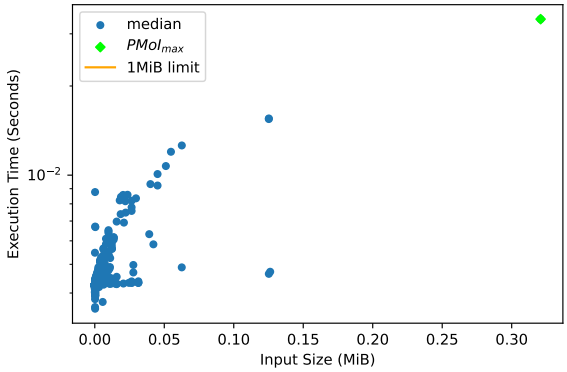
(b) `zlib` $\mathbb{S}1$, Rep 1



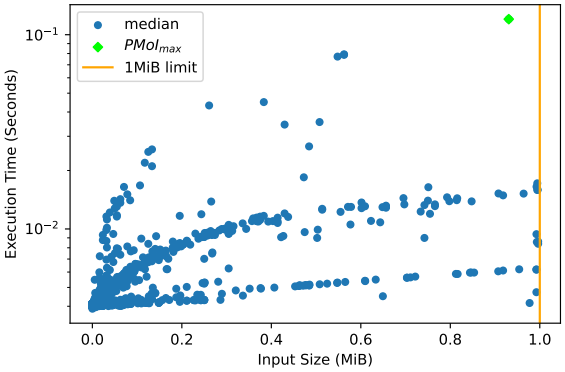
(c) `libjpeg` $\mathbb{S}1$, Rep 1



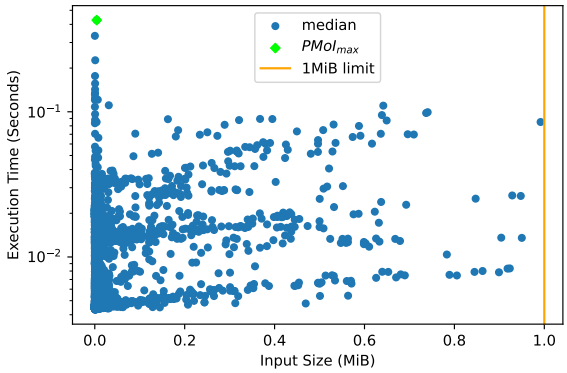
(d) `libxml` $\mathbb{S}1$, Rep 1



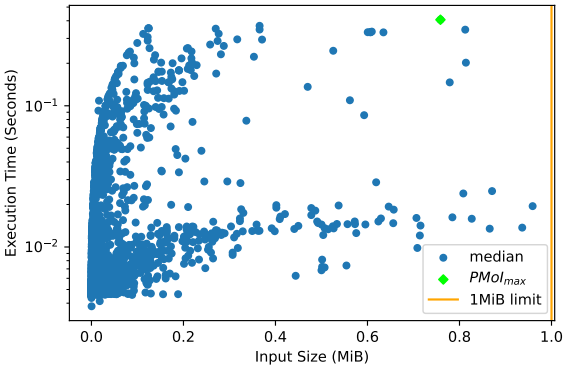
(a) `libpng` $\mathbb{S}1$, Rep 2



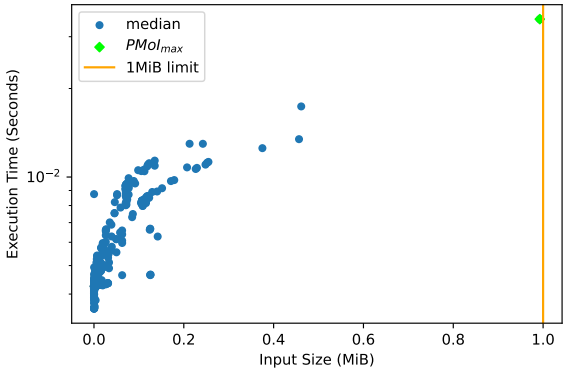
(b) `zlib` $\mathbb{S}1$, Rep 2



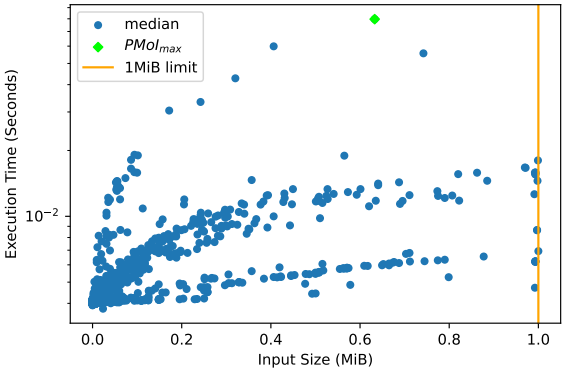
(c) `libjpeg` $\mathbb{S}1$, Rep 2



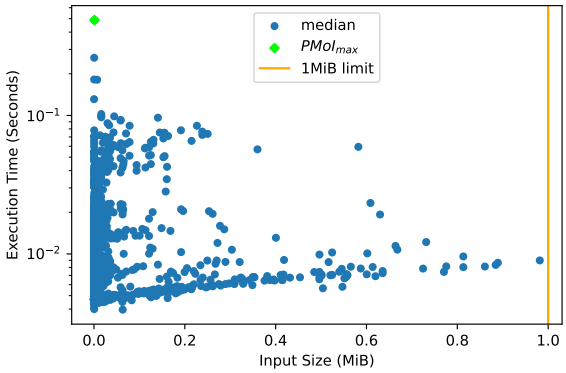
(d) `libxml` $\mathbb{S}1$, Rep 2



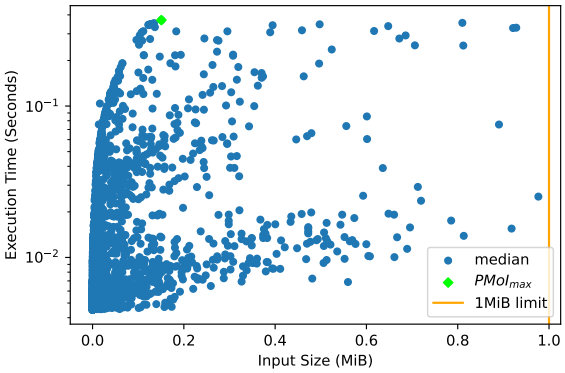
(a) `libpng` $\mathbb{S}1$, Rep 3



(b) `zlib` $\mathbb{S}1$, Rep 3

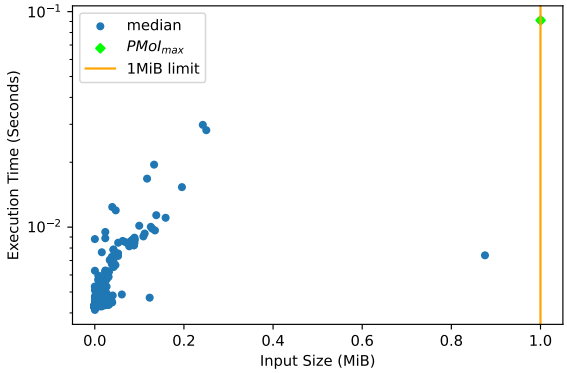


(c) `libjpeg` $\mathbb{S}1$, Rep 3

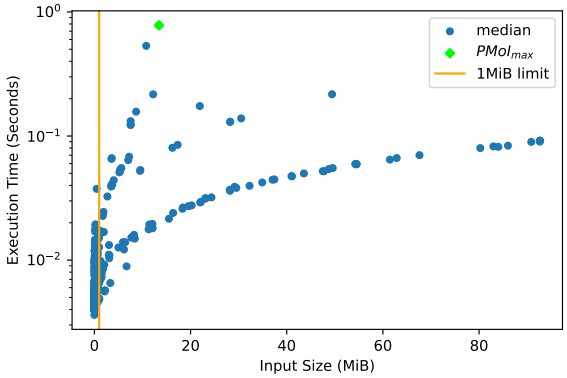


(d) `libxml` $\mathbb{S}1$, Rep 3

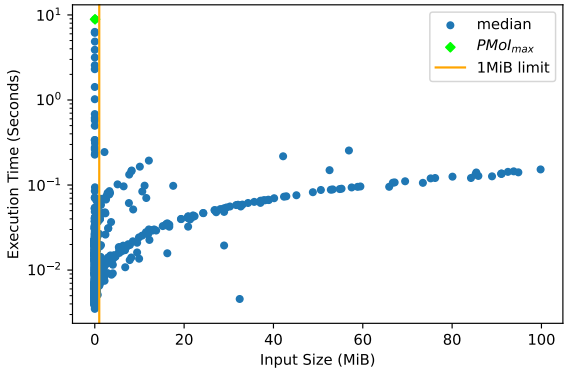
B.2 𝕊2



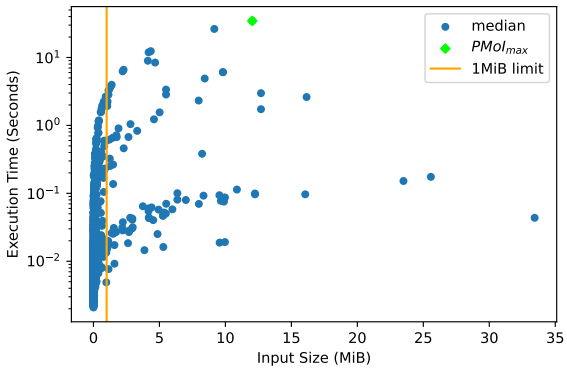
(a) `libpng` 𝕊2, Rep 1



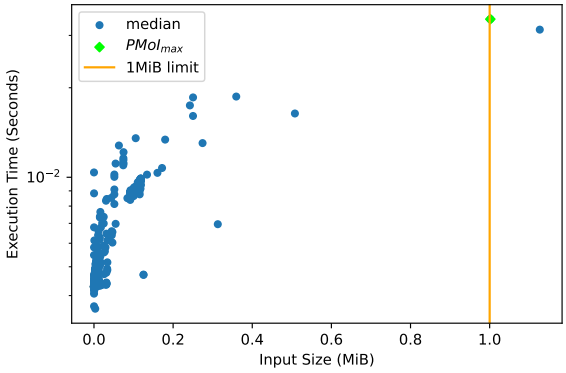
(b) `zlib` 𝕊2, Rep 1



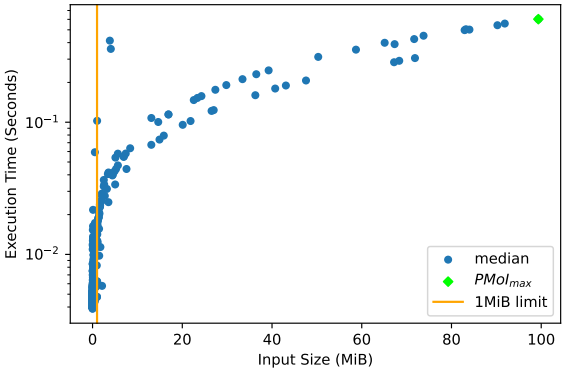
(c) `libjpeg` 𝕊2, Rep 1



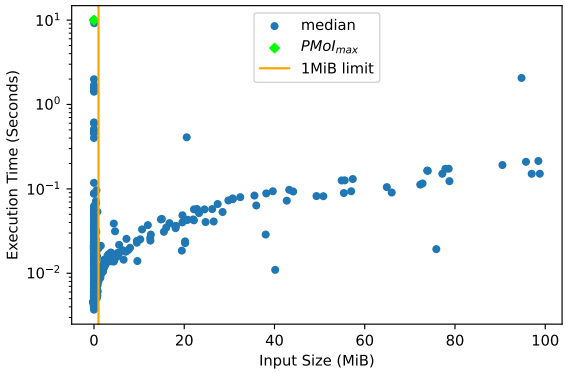
(d) `libxml` 𝕊2, Rep 1



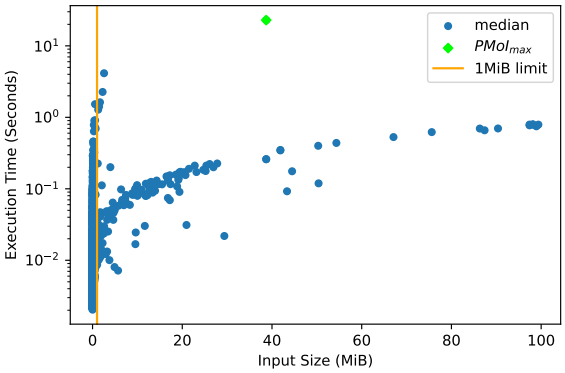
(a) `libpng` $\mathbb{S}2$, Rep 2



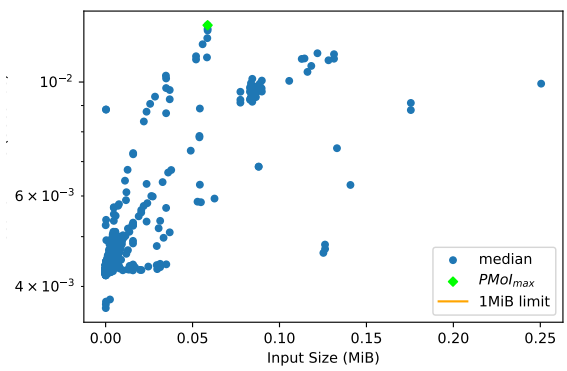
(b) `zlib` $\mathbb{S}2$, Rep 2



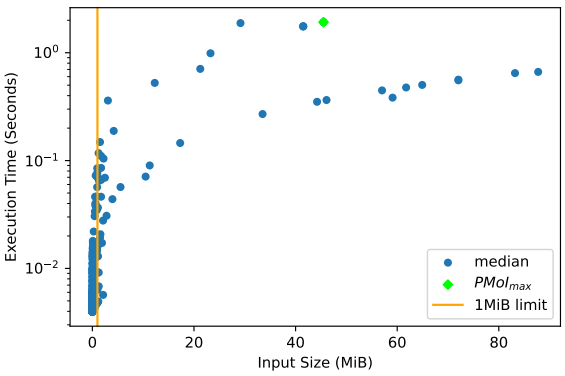
(c) `libjpeg` $\mathbb{S}2$, Rep 2



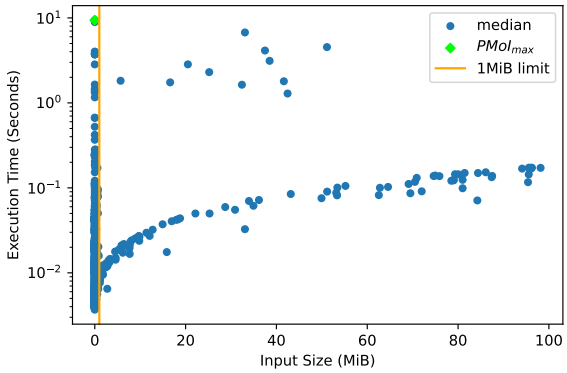
(d) `libxml` $\mathbb{S}2$, Rep 2



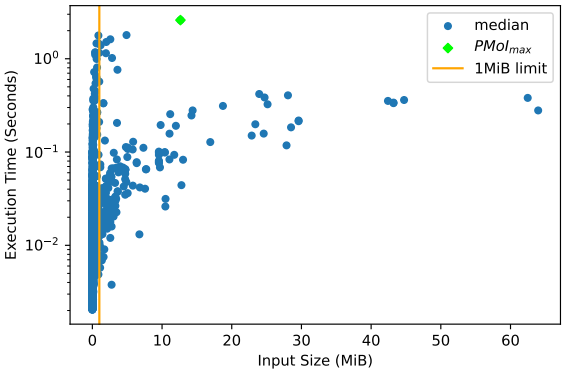
(a) `libpng` $\mathbb{S}2$, Rep 3



(b) `zlib` $\mathbb{S}2$, Rep 3

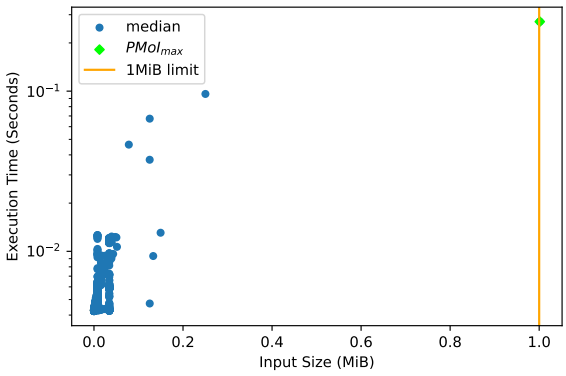


(c) `libjpeg` $\mathbb{S}2$, Rep 3

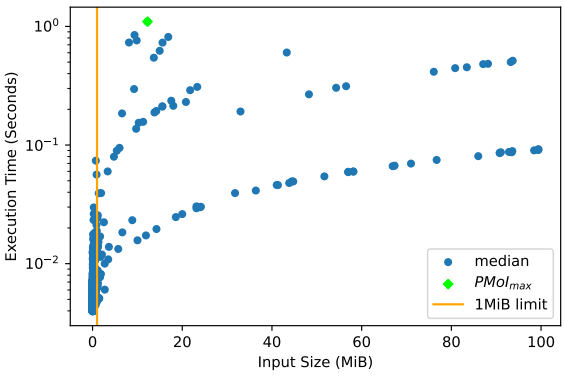


(d) `libxml` $\mathbb{S}2$, Rep 3

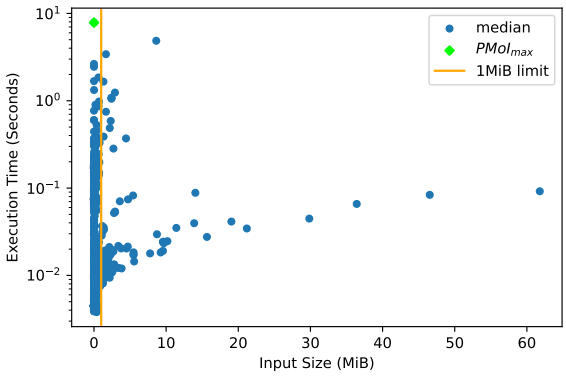
B.3 𝕊3



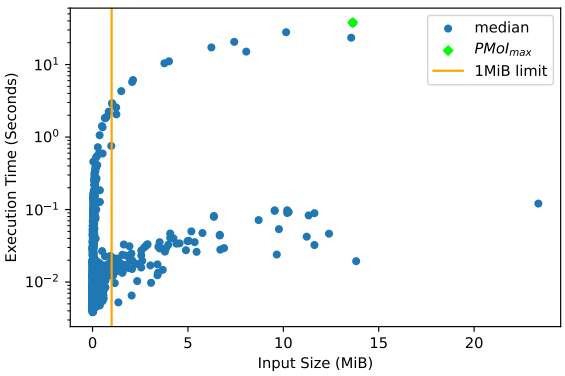
(a) `libpng` 𝕊3, Rep 1



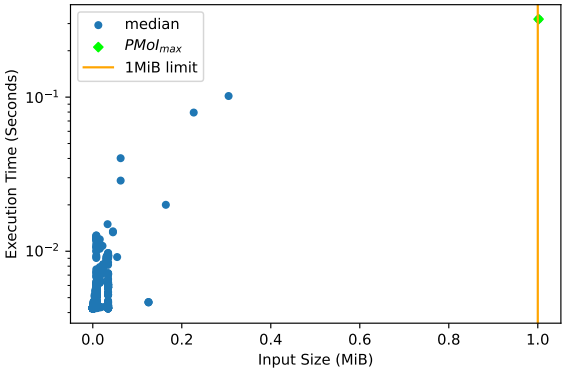
(b) `zlib` 𝕊3, Rep 1



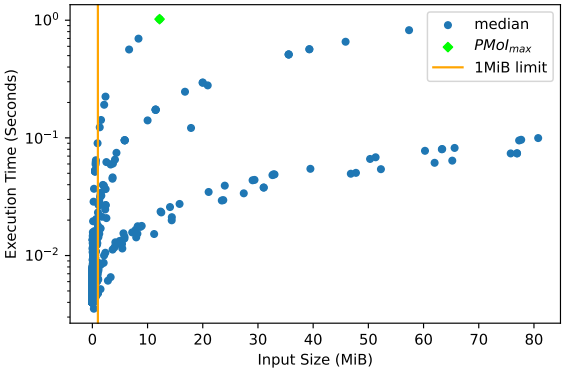
(c) `libjpeg` 𝕊3, Rep 1



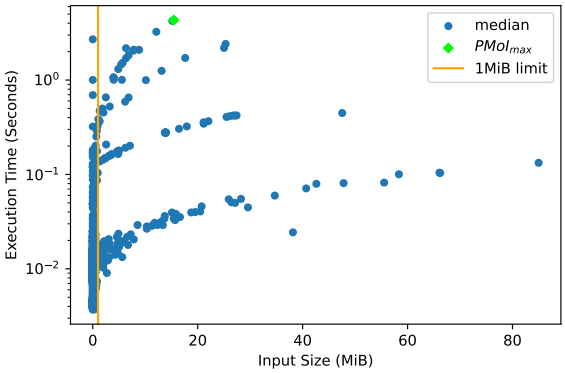
(d) `libxml` 𝕊3, Rep 1



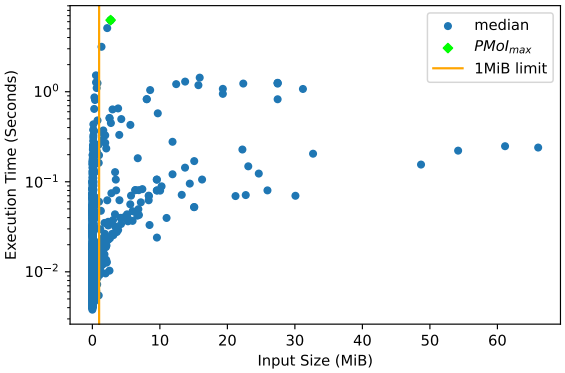
(a) `libpng` 𝕊3, Rep 2



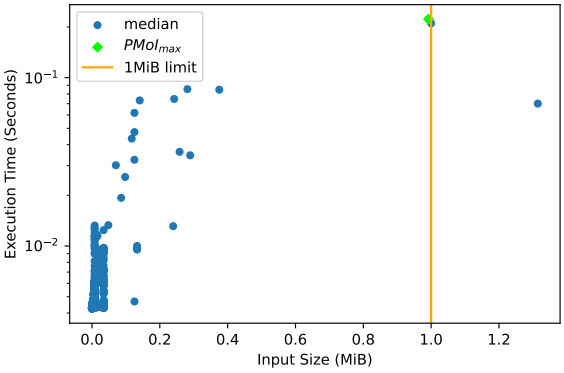
(b) `zlib` 𝕊3, Rep 2



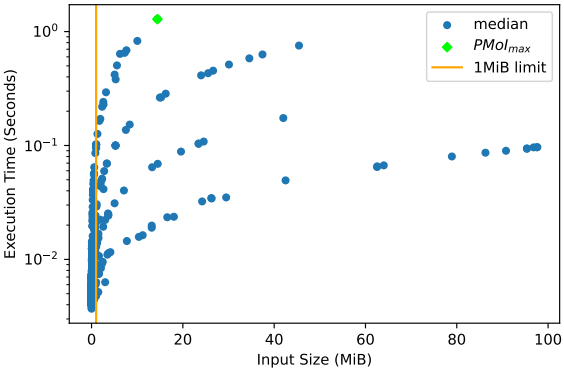
(c) `libjpeg` 𝕊3, Rep 2



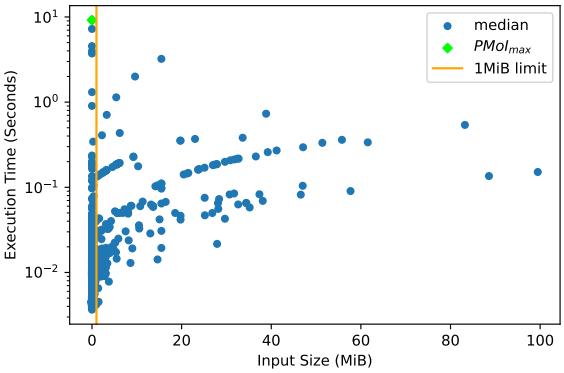
(d) `libxml` 𝕊3, Rep 2



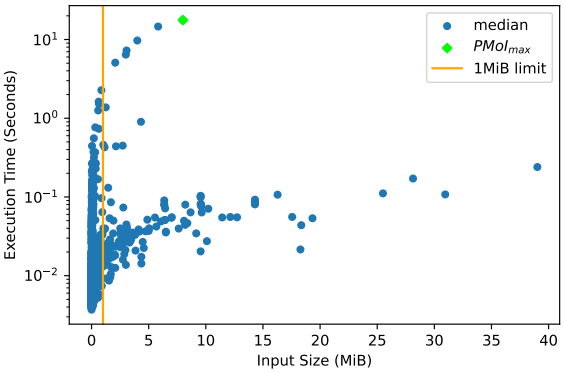
(a) `libpng` $\mathbb{S}3$, Rep 3



(b) `zlib` $\mathbb{S}3$, Rep 3

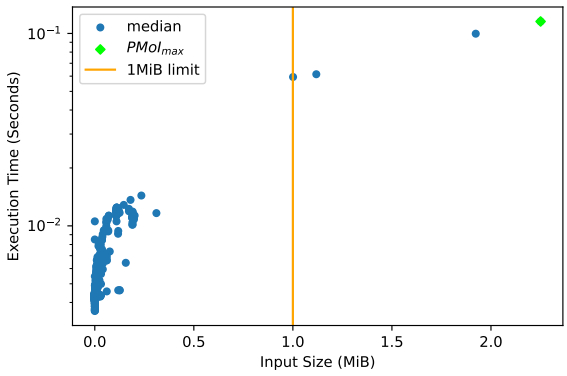


(c) `libjpeg` $\mathbb{S}3$, Rep 3

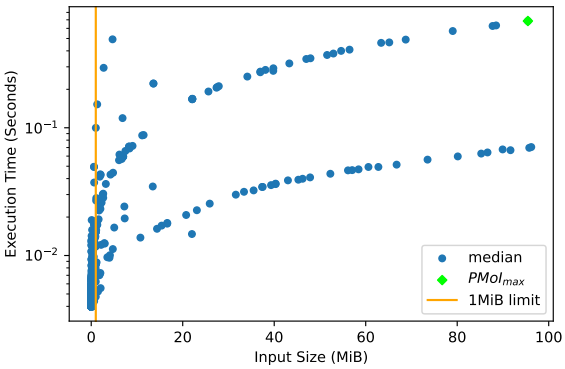


(d) `libxml` $\mathbb{S}3$, Rep 3

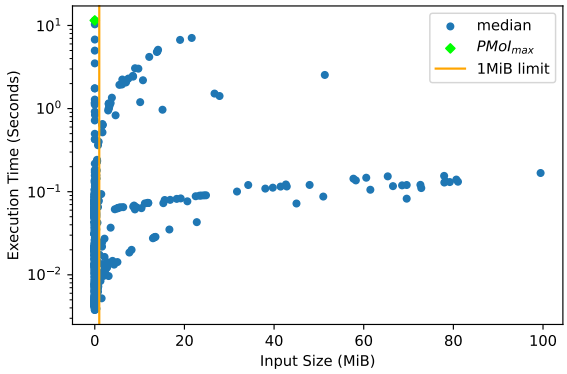
B.4 𝕊4



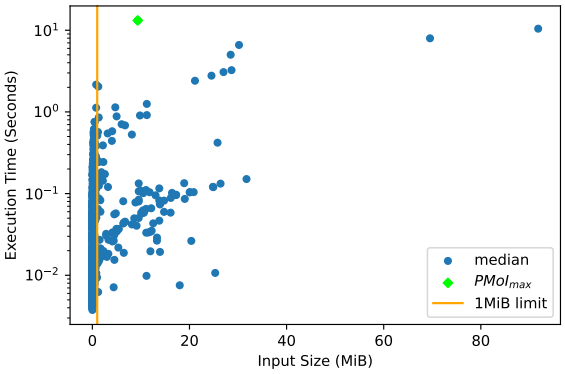
(a) `libpng` 𝕊4, Rep 1



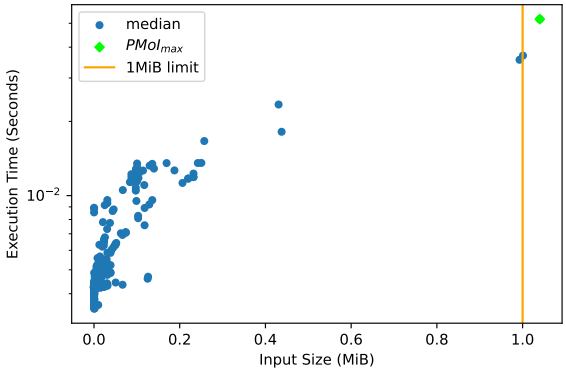
(b) `zlib` 𝕊4, Rep 1



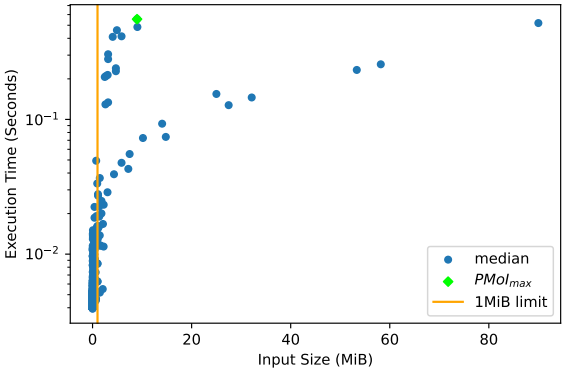
(c) `libjpeg` 𝕊4, Rep 1



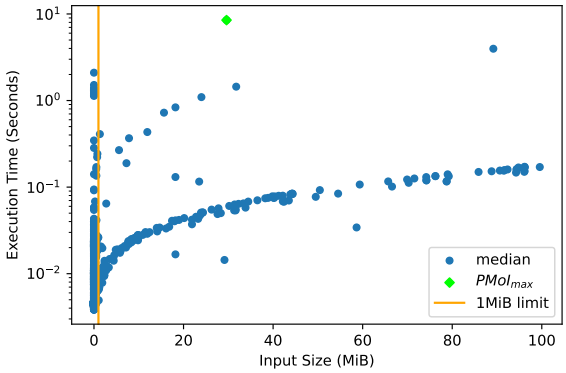
(d) `libxml` 𝕊4, Rep 1



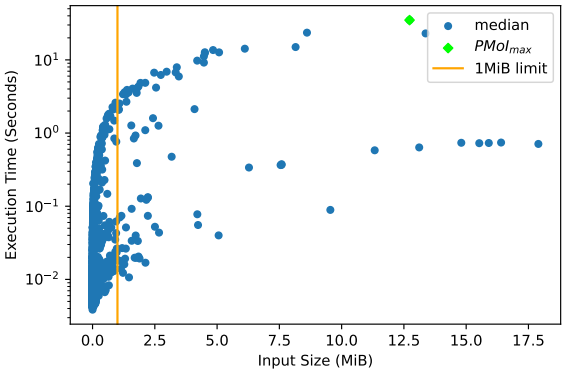
(a) `libpng` $\mathbb{S}4$, Rep 2



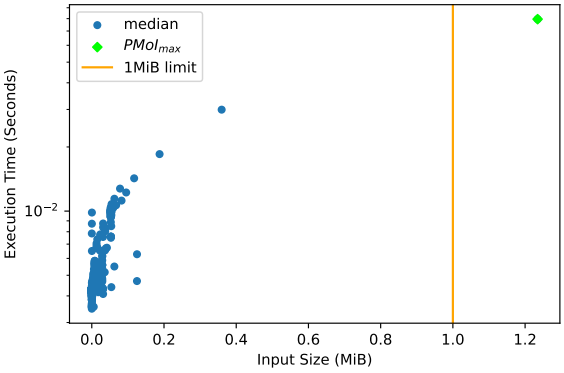
(b) `zlib` $\mathbb{S}4$, Rep 2



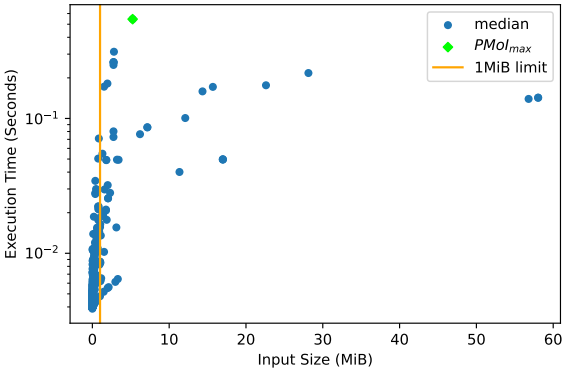
(c) `libjpeg` $\mathbb{S}4$, Rep 2



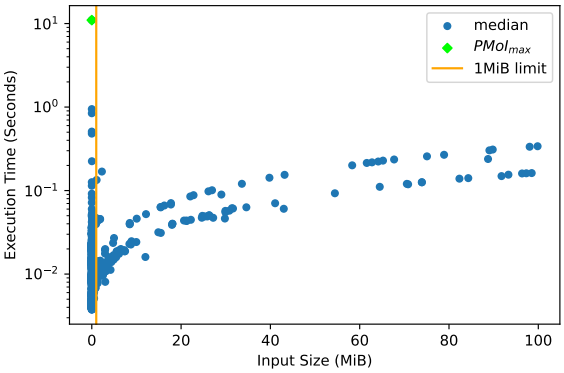
(d) `libxml` $\mathbb{S}4$, Rep 2



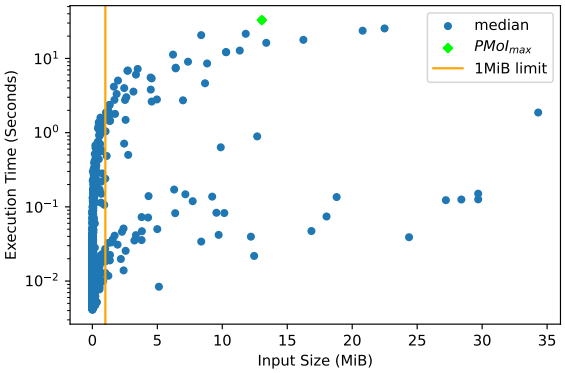
(a) `libpng` 𝕊4, Rep 3



(b) `zlib` 𝕊4, Rep 3

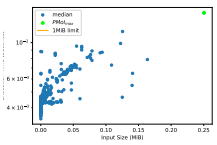


(c) `libjpeg` 𝕊4, Rep 3

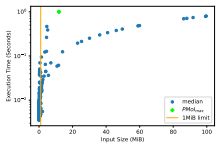


(d) `libxml` 𝕊4, Rep 3

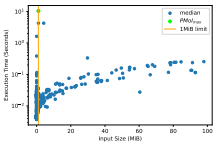
B.5 𝕍1



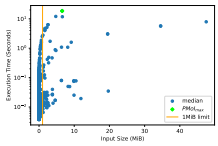
(a) `libpng` 𝕍1, Rep 1



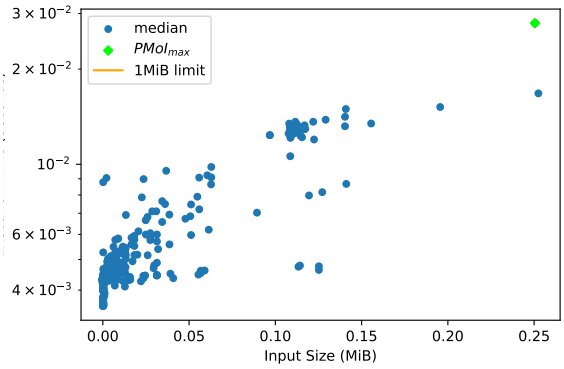
(b) `zlib` 𝕍1, Rep 1



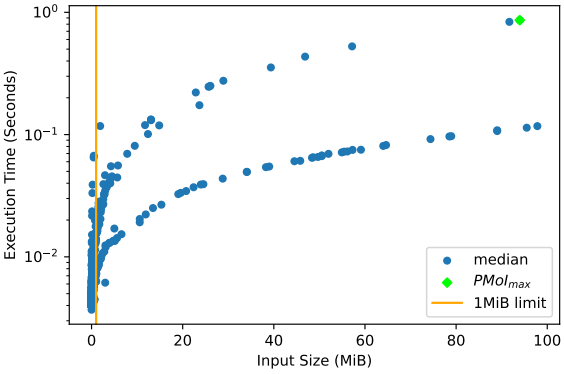
(c) `libjpeg` 𝕍1, Rep 1



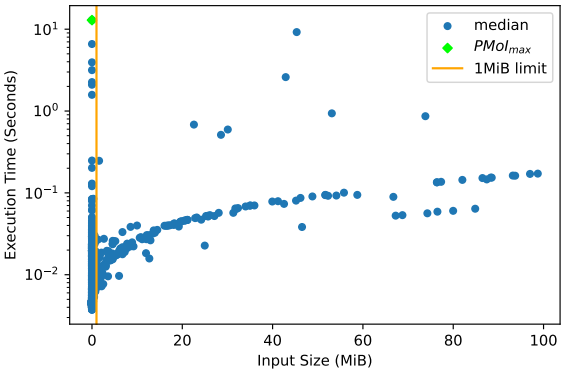
(d) `libxml` 𝕍1, Rep 1



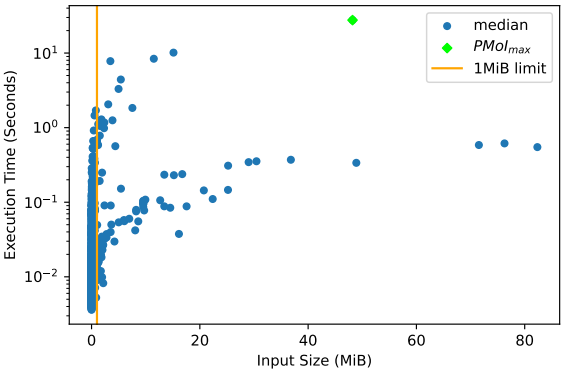
(a) `libpng` $\mathbb{V}1$, Rep 2



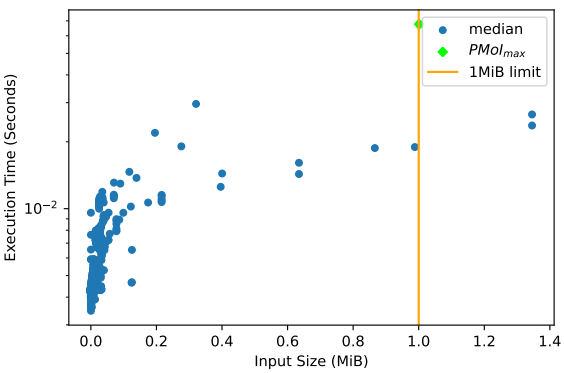
(b) `zlib` $\mathbb{V}1$, Rep 2



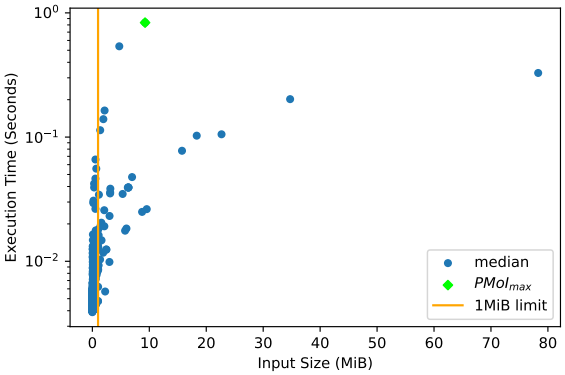
(c) `libjpeg` $\mathbb{V}1$, Rep 2



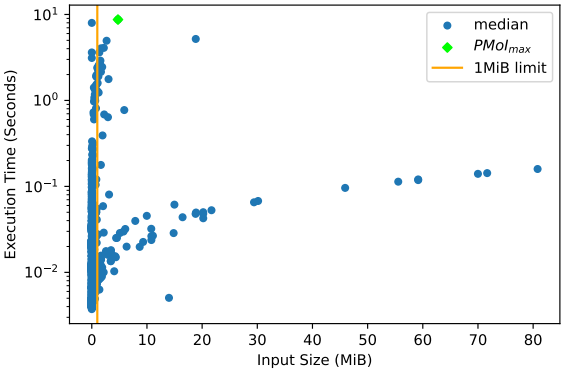
(d) `libxml` $\mathbb{V}1$, Rep 2



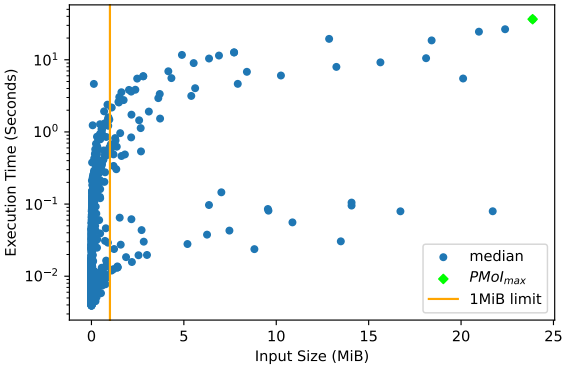
(a) `libpng` 𝕍1, Rep 3



(b) `zlib` 𝕍1, Rep 3

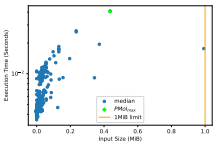


(c) `libjpeg` 𝕍1, Rep 3

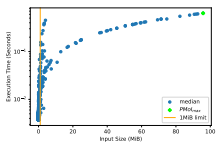


(d) `libxml` 𝕍1, Rep 3

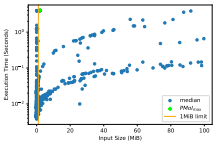
B.6 𝕍2



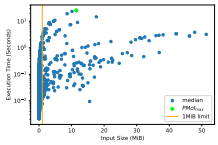
(a) `libpng` 𝕍2, Rep 1



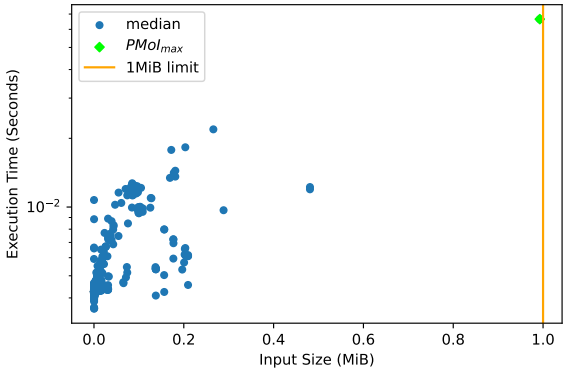
(b) `zlib` 𝕍2, Rep 1



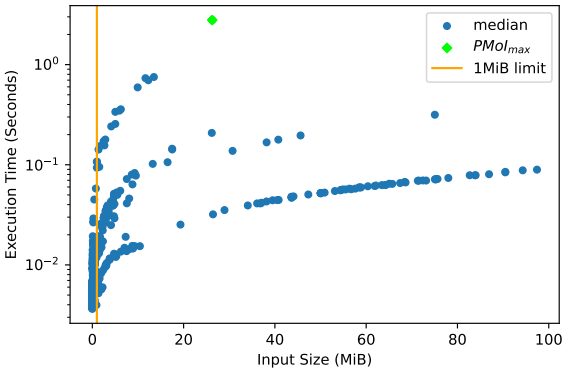
(c) `libjpeg` 𝕍2, Rep 1



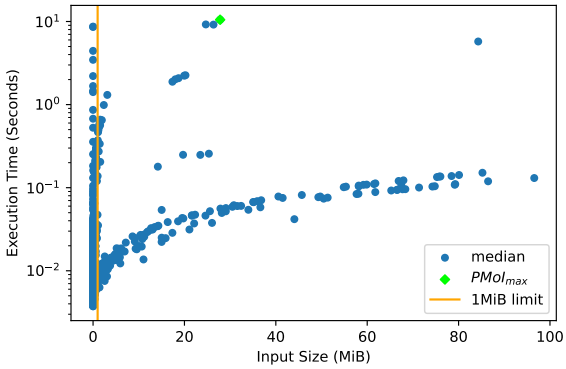
(d) `libxml` 𝕍2, Rep 1



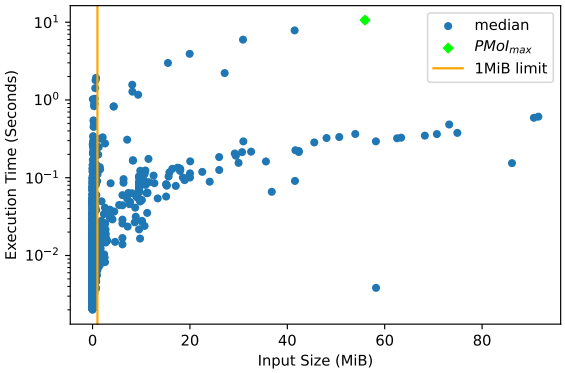
(a) `libpng` $\mathbb{V}2$, Rep 2



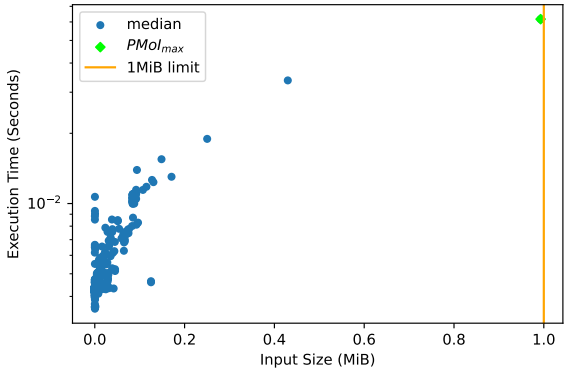
(b) `zlib` $\mathbb{V}2$, Rep 2



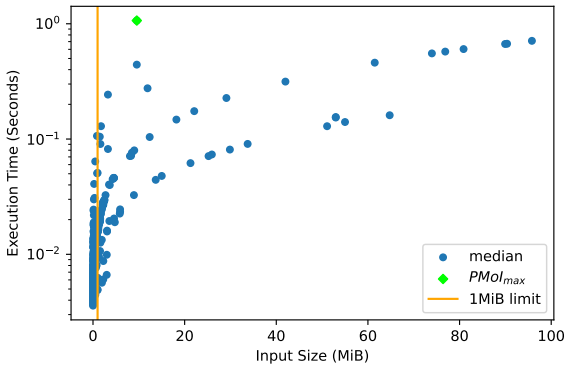
(c) `libjpeg` $\mathbb{V}2$, Rep 2



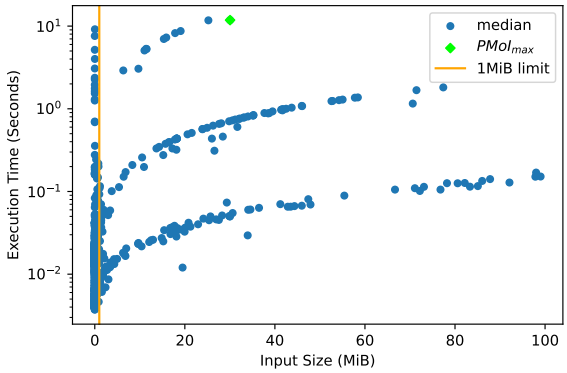
(d) `libxml` $\mathbb{V}2$, Rep 2



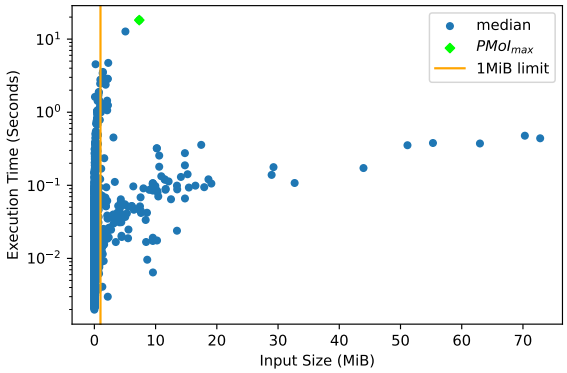
(a) `libpng` 𝕍2, Rep 3



(b) `zlib` 𝕍2, Rep 3



(c) `libjpeg` 𝕍2, Rep 3



(d) `libxml` 𝕍2, Rep 3