Exploring means to facilitate software debugging
SOLTANI, M.S.

Citation
SOLTANI, M. S. (2020, August 25). *Exploring means to facilitate software debugging*. Retrieved from https://hdl.handle.net/1887/135948

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/135948

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/135948 holds various files of this Leiden University dissertation.

Author: Soltani, M.S.
Title: Exploring means to facilitate software debugging
Issue Date: 2020-08-25

Exploring Means to Facilitate Software Debugging

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 25 Augustus 2020
klokke 16:15 uur

door

Mozhan Soltani

geboren te Tehran, Iran
in 1989

Promotiecommissie

Promotor:	Prof. Dr. Thomas Bäck
Copromotors:	Dr. Felienne Hermans
	Dr. Mike Preuss
Promotiecommissie:	Prof. Dr. Tanja Vos (Open University)
	Dr. Claire Le Goues (Carnegie Mellon University)
	Dr. Hadi Hemmati (University of Calgary)
	Prof. Dr. Aske Plaat (voorzitter)
	Prof. Dr. Joost Visser (secretaris)

Contents

1 Introduction **7**
1.1 Automated Test Generation Techniques 9
1.2 Automated Debugging Techniques 10
1.3 Contributions 11
1.3.1 Overview 12
1.3.2 Motivation for the Contributions and their Outcomes 13
1.4 Publications 15

2 Evolutionary Crash Reproduction **17**
2.1 Introduction 17
2.2 Background and Related Work 20
2.2.1 Automated Approaches to Crash Replication 20
2.2.2 Search-based Software Testing 23
2.2.2.1 Genetic Algorithms 24
2.2.3 Unit Test Generation Tools 25
2.2.4 User Studies in Testing and Debugging 26
2.3 The EvoCrash Approach 28
2.3.1 Crash Stack Trace Processing 28
2.3.2 Fitness Function 29
2.3.3 Guided Genetic Algorithm 34
2.3.4 Mocking Strategies 37
2.4 Study I: Effectiveness 38
2.4.1 Research Questions 38

2.4.2 Definition and Context . . . 40
2.4.3 Experimental Procedure . . . 43
2.4.4 Comparison with Coverage-Based Test Generation . . . 44
2.4.5 Crash Reproduction Effectiveness . . . 46
2.4.6 Comparison to State of the Art . . . 50
2.4.7 Threats to Validity . . . 55
2.5 Study II: Usefulness for Debugging . . . 56
2.5.1 Task Selection . . . 56
2.5.2 Experiment Participants . . . 57
2.5.3 Experiment Procedure . . . 58
2.5.4 Data Analysis . . . 60
2.5.5 Statistical Analysis . . . 60
2.5.6 Analysis of the Results . . . 61
2.5.6.1 RQ_4: Impact of EvoCrash Tests on Locating Defects . . . 62
2.5.6.2 RQ_5: Impact of EvoCrash Tests on Fixing Defects . . . 62
2.5.6.3 RQ_6: Impact of EvoCrash Tests on Debugging Time . . . 63
2.5.7 Threats to Validity . . . 64
2.6 Discussion and Lessons Learnt . . . 66
2.7 Conclusions . . . 67

3 Large-scale Evaluation of EvoCrash 73
3.1 Introduction . . . 73
3.2 Background and related work . . . 76
3.2.1 Crash reproduction . . . 76
3.2.2 Search-based crash reproduction with EvoCrash . . . 78
3.2.2.1 Guided genetic algorithm . . . 78
3.2.2.2 Comparison with the state-of-the-art . . . 80
3.3 Benchmark design . . . 80
3.3.1 Projects selection protocol . . . 81
3.3.2 Stack trace collection and preprocessing . . . 83
3.4 The JCrashPack benchmark . . . 84
3.5 Running experiments with ExRunner . . . 85
3.6 Application to EvoCrash: setup . . . 88
3.6.1 Evaluation setup . . . 89
3.7 Application to EvoCrash: results . . . 90
3.7.1 Crash Reproduction Outcomes (RQ1) . . . 91
3.7.1.1 Frames Reproduction Outcomes . . . 92
3.7.1.2 Defects4J applications . . . 93
3.7.1.3 XWiki and Elasticsearch . . . 93

3.7.1.4 Exceptions . . . 95
3.7.2 Impact of Exception Type and Project on Performance (RQ2) . . 95
3.8 Challenges for crash reproduction (RQ3) . . . 99
3.8.1 Input data generation . . . 99
3.8.2 Complex code . . . 101
3.8.3 Environmental dependencies . . . 102
3.8.4 Static initialization . . . 103
3.8.5 Abstract classes and methods . . . 104
3.8.6 Anonymous classes . . . 104
3.8.7 Private inner classes . . . 105
3.8.8 Interfaces . . . 105
3.8.9 Nested private calls . . . 105
3.8.10 Empty `enum` type . . . 106
3.8.11 Frames with `try/catch` . . . 106
3.8.12 Missing line number . . . 107
3.8.13 Incorrect line numbers . . . 108
3.8.14 Unknown . . . 108
3.9 Discussion . . . 109
3.9.1 Empirical evaluation for crash reproduction . . . 109
3.9.2 Usefulness for debugging . . . 110
3.9.3 Benchmark building . . . 111
3.10 Future research directions for search-based crash reproduction . . . 112
3.10.1 Context matters . . . 112
3.10.2 Stack trace preprocessing and target frame selection . . . 113
3.10.3 Guided search . . . 113
3.10.4 Improving testability . . . 114
3.11 Threats to validity . . . 114
3.12 Conclusion . . . 115

4 Fitness Function Evaluation 123
4.1 Introduction . . . 124
4.2 Background and Related Work . . . 125
4.2.1 Related Work . . . 126
4.2.2 EvoCrash . . . 126
4.2.2.1 Weighted Sum (WS) Fitness Function . . . 126
4.2.2.2 Guided Genetic Algorithm (GGA) . . . 127
4.3 Single-Objective and Multi-Objectivization for Crash Reproduction . . . 128
4.3.1 Constraints Relaxation . . . 128
4.3.2 Multi-objectivization . . . 129

4.3.3 Graphical Interpretation . . . 131
4.4 Empirical Evaluation . . . 132
4.4.1 Setup . . . 132
4.4.2 Analysis . . . 133
4.5 Results . . . 134
4.6 Discussion . . . 135
4.7 Conclusion . . . 137

5 The Significance of Bug Report Elements **141**
5.1 Introduction . . . 141
5.2 Research Methodology . . . 144
5.2.1 Interviews . . . 145
5.2.1.1 Protocol . . . 146
5.2.1.2 Participants . . . 146
5.2.1.3 Data Analysis . . . 146
5.2.2 Surveying Developers . . . 147
5.2.2.1 protocol . . . 147
5.2.2.2 Participants . . . 148
5.2.2.3 Data Analysis . . . 148
5.2.3 Mining Github Issues . . . 148
5.2.3.1 Analysis of the Mined Issues . . . 149
5.3 The *IMaChecker* Approach . . . 150
5.4 Results . . . 153
5.4.1 **RQ$_1$.** What types of information do developers perceive as important in bug reports? . . . 154
5.4.2 **RQ$_2$.** Do the important elements in bug reports impact bug resolution times? . . . 155
5.4.3 **RQ$_3$.** How often do bug reports contain the important elements? 163
5.5 Discussion . . . 164
5.5.1 Bug Report Templates and User Support . . . 164
5.5.2 Representative Samples . . . 165
5.5.3 Internal Validity of the Experiments . . . 165
5.5.4 Generalizability of Results . . . 166
5.5.5 Automated Crash Reproduction . . . 167
5.5.6 What Do User Contents Provide? . . . 167
5.6 Related Work . . . 167
5.7 Conclusions . . . 169

6 The Use of Contracts in Open Source Software **185**
6.1 Introduction . . . 185

6.2 Related Work . . . 188
6.2.1 Assertion Use and Impact on Quality . . . 188
6.2.2 Empirical Studies on Github Projects . . . 189
6.3 Research Methodology . . . 190
6.3.1 Automated Contract Detection . . . 191
6.3.2 Parsing Commit Logs . . . 196
6.4 Results . . . 199
6.4.1 **RQ_1. How often are different types of contracts used?** . . . 200
6.4.2 **RQ_2. For which use cases do developers use contracts?** . . . 201
6.4.3 **RQ_3. Does the use of contracts relate to occurrence of defects?** 203
6.5 Discussion . . . 203
6.5.1 Preference for Different Contracts . . . 203
6.5.2 Automated Semantic Analysis of Contracts . . . 204
6.5.3 Effect of Using Contracts . . . 205
6.6 Threats to Validity . . . 205
6.6.1 Threats to Internal Validity . . . 205
6.6.2 Generalizability of Findings . . . 206
6.7 Conclusion . . . 206

7 Dutch Summary **209**

8 English Summary **211**

9 About the Author **213**

Bibliography **215**

1 Introduction

Today computer systems have profound impact on every aspect of our lives. Software applications have transformed education systems, healthcare, military systems, the way businesses operate, the way individuals gather information and communicate, and so on. Thus, in numerous ways, we have been relying on computer systems to the extent it is difficult to imagine daily life without them. Despite the advantages of using computer systems, over the last decades, catastrophic cases of their failures have been reported.

(a) Lifting off

(b) Exploding

Figure 1.1: The lift-off and explosion of the Ariane 5 rocket in 1996.

As Figure 1.1 shows, the explosion of the Ariane 5 rocket in 1996 is among the older and well-known failures which was caused due a single integer overflow. A more recent case was reported in 2015 in which a robot killed a worker at one of the production plants of Volkswagen in Germany [174]. In this case the robot took the

Figure 1.2: Grounding the entire American Airline fleet [39].

Figure 1.3: The twit from Facebook confirming the outages of Facebook, Instagram, and WhatsApp applications.

worker and crushed him against a metal plate. Even more recent cases from 2018 and 2019 are reported [39, 131] which range from the outages of Facebook, Instagram, and WhatsApp (shown in Figure 1.3), separating Amazon users from their cloud storage, grounding the entire American Airline fleet (shown in Figure 1.2), canceling over 100 flights by British Airways, pushing the investment firm Knight Capital into bankruptcy, and so on. These failures disrupted lives on a large scale, caused deaths, and cost billions of dollars.

In order to prevent such failures and their disastrous consequences, a large number of quality and functional requirements must be met. Such requirements, in addition to complexity of software products, and short software development cycles

place major importance on activities in the areas of software verification, testing, and debugging [116]. In a typical commercial development organization, the cost of software verification, testing and debugging activities can easily range from 50 to 75 percent of the total development cost [116]. In addition, these activities are error-prone and labor-intensive. Therefore, much research effort has been put into automating as many activities in these areas as possible [105]. In what follows, we present a background about the automated techniques which are researched in the areas of test generation, and software debugging. Thereafter, we present the contributions in this thesis, and conclude the chapter with a list of peer-reviewed publications associated with each of the following chapters in this thesis.

1.1 Automated Test Generation Techniques

Various automated techniques for unit test generation have been proposed in the research literature. DART [112] is an older technique which combines directed testing with random testing to automatically generate test cases. PEX [206] is also an older white-box test generation tool which uses dynamic symbolic execution to generate tests for .Net programs.

Recent test generation approaches include SAPIENZ [159], EvoSuite [103], and JTExpert [196]. SAPIENZ [159] is an approach to Android testing that uses multi-objective search to automatically optimize test sequences, by minimizing test lengths, while simultaneously maximizing coverage and fault revelation. EvoSuite [103] generates test suites for Java classes. To this end, Evosuite applies a hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior. JTExpert [196], on the other hand, uses a static analysis approach to extract the methods or constructors which change the state of the class under test or that may reach a test target. Then JTExpert [196] uses a generator to make instances of classes by using subclasses and external factory methods. JTExpert also uses a seeding strategy and a diversification strategy to increase the likelihood to reach a test target.

Moreover, research literature also contains the Monkey [18], Dynodroid [156], T3 [185], and RANDOOP [177] techniques which apply various random test generation approaches. Monkey is a command line tool that can be run on any instance of Android emulator or device to stress-test applications. This tool generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events [18]. Dynodroid [156] views an Android application as an

event-driven program that interacts with its environment by means of a sequence of events through the Android framework. By instrumenting the framework once, Dynodroid monitors the reaction of an application upon each event to guide the generation of the next events. Dynodroid also allows interleaving events from machines, which are better at generating a large number of simple inputs, with events from humans, who are better at providing intelligent inputs.

Pacheco et al. [177] propose RANDOOP, which incorporates feedback from executing test inputs as they are created. This technique builds inputs incrementally by randomly selecting a method call to apply and finding arguments from previously constructed inputs. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test.

Finally, T3 [185] is the next generation of the testing tool T2 [186]. T3 is implemented in Java 8, and can be used to automatically generate random tests for Java classes. A test sequence against a Class Under Test (CUT) starts with the creation of an object which is an instance of the CUT, followed by calls to the methods of the object, or updates to the fields. T3 randomly generates a large amount of such test sequences to trigger faulty behavior, and thus finding a bug.

1.2 Automated Debugging Techniques

Andreas Zeller proposes the delta debugging algorithm [220] which systematically narrows the state difference between a passing run and a failing run, to determine whether a change in the program state makes a difference in the test outcome. Applying the delta debugging algorithm to multiple states of the program automatically reveals the cause-effect chain of the failure, which are, the variables and values that caused the failure [220]. Slicing programs is another automated technique described by Andreas Zeller [221], in which, based on statement dependencies, one can focus on specific subsets of the program, which are called slices. These subsets may have influenced a specific statement or be influenced by a specific statement. Therefore, in this approach, depending on the direction of computing the dependencies to or from a statement, forward or backward slicing operations are performed.

Parsa et al. [182] propose an approach to defect localization using elastic net. The proposed approach finds the smallest effective subset of program predicates known as bug predictors. After selecting bug predictors, the main causes of faults are detec-

ted by using existing program slicing techniques. Eichinger et al. [96] propose another approach to automated defect localization using dataflow-enabled call graphs that incorporate abstractions of the dataflow. In this approach, defect localization is essentially formulated as a data mining problem, making use of discretisation, frequent subgraph mining and feature selection [96]. Moreover, another approach to fault localization is proposed by Naish et al. [168]. This approach is based on ranking program statements or blocks according to how likely they are to be buggy.

Chandra et al. [80] propose angelic debugging. This technique is based on locating expressions that are likely to be bugs. After locating the likely-buggy expressions, angelic debugging searches the space of all edits to the program for one that repairs the failing test without breaking any passing test. Furthermore, Wei et al. [210] propose the AutoFix-E approach which automatically generates and validates fixes for software faults. AutoFix-E relies on contracts present in the software to ensure that the proposed fixes are semantically sound. On the other hand, Le Goues et al. [148] propose GenProg, which uses an extended form of genetic programming to repair defects in off-the-shelf, legacy programs without formal specifications, program annotations, or special coding practices. GenProg evolves program variants which keep required functionality but are not susceptible to given defects, using existing test suites.

Another approach to assist with software debugging is proposed by Zhang et al. [223] to automatically identify breakpoints. This technique combines the nearest neighbor queries method, dynamic program slicing, and memory graph comparison to detect suspicious program statements and states. Based on this information, breakpoints are generated and divided into two groups, where the primary group contains conditional breakpoints and the secondary group contains unconditional ones.

Moreover, a body of research is done on automated crash reproduction. Earlier research in this field [58,83,169,205] use record-replay techniques to reproduce crashes. However, due to the execution overhead and privacy-related issues, recent research work [81,137,150,151,171,194,215,217,219] take a post-failure approach by using the crash data that is generated after the crash occurs. The crash data is often in the form of crash stack traces which are recorded in software execution logs.

1.3 Contributions

In this section, first we present an overview of the overarching goal of the thesis and an outline of the research questions we aimed to answer. Thereafter, we provide our motivation for each of the contributions in this thesis, followed by their outcomes.

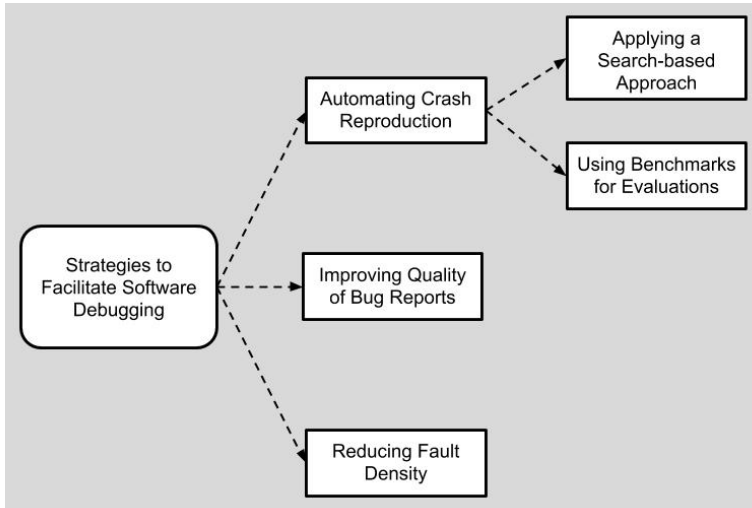


Figure 1.4: The overview of the strategies addressed in this thesis to facilitate software debugging.

1.3.1 Overview

As Figure 1.4 shows, the overarching goal of this thesis is to investigate various means to facilitate automated software debugging. We start with investigating a search-based solution for automated crash reproduction to support developers by producing information which help identify the faults faster. While investigating the effectiveness of this solution for developers, we detect additional needs to address:

- an extensible benchmark for comparing automated crash reproduction techniques, and a systemic way to run evaluations using this benchmark,
- investigating techniques to improve the fitness function in the proposed solution for search-based automated crash reproduction,
- investigating the information developers need from bug reports, which make software debugging more efficient, and
- investigating whether the Design by Contract (DBC) software development approach can reduce defect density in software, so that debugging effort is reduced by trying to prevent faults in software products in the first place.

To address the above concerns, we define the following research questions:

- **RQ_1:** How can search-based algorithms be used to automate crash reproduc-

tion?

- **RQ$_2$:** How to create a benchmark of representative crashes to use for evaluating crash reproduction research prototypes?
- **RQ$_3$:** What is the impact of multi-objectivization on evolutionary crash reproduction?
- **RQ$_4$:** Do different elements of bug reports impact the time it takes to fix bugs?
- **RQ$_5$:** Can program contracts be used to reduce the occurrence of bugs?

1.3.2 Motivation for the Contributions and their Outcomes

As mentioned in Section 1.2 previously, a body of research was done on automated crash reproduction. Despite the previous research in this area, application of evolutionary search-based algorithms remained underexplored. This is while search-based algorithms have proven to be efficient techniques in other areas such as automated test generation.

To evaluate the application of search-based algorithms for automated crash reproduction and answer **RQ$_1$**, we devised a problem representation, a new fitness function, and new genetic algorithm operators, in the first study we performed. We named this approach EvoCrash. To implement EvoCrash, we extended the search based test generating tool, named EvoSuite [103]. In the first study we did, we compared EvoCrash with the state-of-the-art crash reproduction approaches. Our evaluation indicated EvoCrash outperformed the state-of-the-art techniques by reproducing more crashes from real-world projects.

Despite the early evaluations we performed, we noticed the number of projects and available crash stack traces are rather limited, which makes comparison of the research prototypes challenging. Therefore, in the second study we answer **RQ$_2$** by proposing JCrashPack, an extensible benchmark for Java crash reproduction. JCrashPack contains 200 stack traces from various Java projects, including industrial open source ones, on which we run an extensive evaluation of EvoCrash. EvoCrash successfully reproduced 43% of the crashes. Furthermore, we observed that reproducing NullPointerException, IllegalArgumentException, and IllegalStateException is relatively easier than reproducing ClassCastException, ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException.

To answer **RQ$_3$**, in the third study, we continue with further evaluating the EvoCrash approach. In this study, we investigate the alternatives to the original fitness func-

tion we defined in EvoCrash. The original fitness function in EvoCrash uses single weighted sum scalarization function to optimize test generation. Therefore, in the third study, we assessed whether there is any difference if we use a simple sum scalarized function without any weights. In addition, we assessed whether multi-objectivization, which is the process of turning a single-objective optimization to multi-objective optimization, impacts crash reproduction in any way. Our results indicate that for complex crashes the weighted sum function reduces the test case generation time, compared to the simple sum function, while for simpler crashes the effect is the opposite. Similarly, for complex crashes, multi-objectivization reduces test generation time compared to optimizing with the weighted sum function.

When it comes to debugging, the quality of the information that is available to developers can vary. Too little information can influence the priority that is assigned to bug reports. To answer **RQ**$_4$, the fourth study we performed focuses on the quality of bug reports and the significance of bug report elements on bug resolution times. In this study, we interviewed 35 developers and surveyed 305 developers to understand developers' perception on the significance of different bug report elements. According to the results, developers find it highly important that bug reports include crash description, reproducing steps, and crash stack traces. In addition, to evaluate the quality of currently available bug reports, we mined issue repositories of 250 most popular projects on Github. Statistical analysis on the mined issues shows that crash reproducing steps, stack traces, fix suggestions, and user contents, have statistically significant impact on bug resolution times, for ∼70%, ∼76%, ∼55%, and ∼33% of the projects. However, on average, over 70% of bug reports lack these elements.

When it comes to software testing, verification, and debugging with the intention to meet the quality and functional requirements, one aspect to consider is identifying means by which the need for software debugging is reduced. Yuan et al. [218] show in their study that simple testing can prevent most critical failures. Similarly, the studies by Kochhar and Lo [143] and Casalnuovo et al. [77] show there is a negative correlation between the use of program contracts and the frequency of bug occurrence for a given method in the program.

In the fifth study we perform, we extend the studies done by Kochhar and Lo [143] and Casalnuovo et al. [77] to answer **RQ**$_5$. In this study, we report results of an empirical evaluation on the use of contracts in 148 open source projects, written in Java, C++, and Python. Our findings show that the average use of different types of contracts differ depending on the program language. Furthermore, the results of regression analysis shows there is a negative relation between the number of contracts and frequency of defect occurrence in a method. These results are statistically significant for all Java, C++, and Python projects.

1.4 Publications

The different chapters of this thesis are based on the following publications:

- Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. "A guided genetic algorithm for automated crash reproduction." In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 209-220. IEEE, (2017). (Chapter 2)
- Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. "Search-based crash reproduction and its impact on debugging." IEEE Transactions on Software Engineering (2018). (Chapter 2)
- Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie Van Deursen. "A benchmark-based evaluation of search-based crash reproduction." Empirical Software Engineering 25, no. 1, pp. 96-138. Springer, (2020). (*The first two authors agreed to share this paper in their PhD theses.*) (Chapter 3)
- Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. "Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction." In International Symposium on Search Based Software Engineering, pp. 325-340. Springer, Cham, (2018). (Chapter 4)
- Mozhan Soltani, Felienne Hermans, and Thomas Bäck. "The significance of bug report elements." Empirical Software Engineering. Springer, (2020). (*This paper is under revision at the time of writing the thesis.*) (Chapter 5)

2 Evolutionary Crash Reproduction

Software systems fail. These failures are often reported to issue tracking systems, where they are prioritized and assigned to responsible developers to be investigated. When developers debug software, they need to reproduce the reported failure in order to verify whether their fix actually prevents the failure from happening again. Since manually reproducing each failure could be a complex task, several automated techniques have been proposed to tackle this problem. Despite showing advancements in this area, the proposed techniques showed various types of limitations. In this paper, we present EvoCrash, a new approach to automated crash reproduction based on a novel evolutionary algorithm, called Guided Genetic Algorithm (GGA). We report on our empirical study on using EvoCrash to reproduce 54 real-world crashes, as well as the results of a controlled experiment, involving human participants, to assess the impact of EvoCrash tests in debugging. Based on our results, EvoCrash outperforms state-of-the-art techniques in crash reproduction and uncovers failures that are undetected by classical coverage-based unit test generation tools. In addition, we observed that using EvoCrash helps developers provide fixes more often and take less time when debugging, compared to developers debugging and fixing code without using EvoCrash tests.

2.1 Introduction

Despite the significant effort spent by developers in software testing and verification, software systems still fail. These failures are reported to issue tracking systems, where

they are prioritized, and assigned to responsible developers for inspection. When developers debug software, they need to reproduce the reported failure, understand its root cause, and provide a proper fix that prevents the failure. While crash stack traces indicate the type of crash and the method calls executed at the time of the crash, they may lack critical details that a developer could use to debug the software. Therefore, depending on the complexity of the reported failures and amount of available information about them, manual crash reproduction can be a labor-intensive task which negatively affects developers' productivity.

To reduce debugging effort, researchers have proposed various automated techniques to generate test cases reproducing the target crashes. Generated tests can help developers better understanding the cause of the crash by providing the input values that actually induce the failure and enable the usage of a debugger in the IDE with runtime data. To generate such tests, crash reproduction techniques leverage various sources of information, such as stack traces, core dumps, failure descriptions. As Chen and Kim [81] first identified, these techniques can be classified into two categories: record-replay techniques, and post-failure techniques. Record-replay approaches [**?**, **?**, 58, 169, 205] monitor software behavior via software/hardware instrumentation to collect the observed objects and method calls when failures occur. Unfortunately, such techniques suffer from well-known practical limitations, such as performance overhead [81], and privacy issues [171].

As opposed to these costly techniques, *post-failure* approaches [81, 150, 151, 171, 194, 215, 219] try to replicate crashes by exploiting data that is available *after* the failure, typically stored in log files or external bug tracking systems. Most of these techniques require specific input data in addition to crash stack traces [81], such as core dumps [150, 151, 194, 208] or software models like input grammars [136, 137] or class invariants [69].

Since such additional information is usually not available to developers, recent advances in the field have focused on crash stack traces as the *only* source of information for debugging [81, 171, 215]. For example, Chen and Kim developed STAR [81], an approach based on backward symbolic execution that outperforms earlier crash replication techniques, such as Randoop [177] and BugRedux [134]. Xuan et al. [215] presented MuCrash, a tool that mutates existing test cases using specific operators, thus creating a new pool of tests to run against the software under analysis. Nayrolle et al. [171] proposed JCHARMING, based on directed model checking combined with program slicing [171, 172].

Unfortunately, the state-of-the-art tools suffer from several limitations. For example, STAR cannot handle cases with external environmental dependencies [81] (e.g., file

or network inputs), non-trivial string constraints, or complex logic potentially leading to a path explosion. MuCrash is limited by the ability of existing tests in covering method call sequences of interest, and it may lead to a large number of unnecessary mutated test cases [215]. JCHARMING [171,172] applies model checking which can be computationally expensive. Moreover, similar to STAR, JCHARMING does not handle crash cases with environmental dependencies.

This paper is an extension of our previous conference paper [203], where we presented EvoCrash, a search-based approach for the automated crash replication problem and built on top of EvoSuite [103], which is a well-known coverage-based unit test generator for Java code. Specifically, EvoCrash uses a novel evolutionary algorithm, namely Guided Genetic Algorithm (GGA), which leverages the stack trace to guide the search toward generating tests able to trigger the target crashes. GGA uses a generative routine to build an initial population of test cases, which exercise at least one of the methods reported in the crash stack frames (target methods). GGA also uses two novel genetic operators, i.e., namely *guided crossover* and *guided mutation*, to ensure that the test cases keep exercising the target methods across the generations. The search is further guided by a fitness function that combines coverage-based heuristics with a crash-based heuristic measuring the distance between the stack traces (if any) generated by the candidate test cases and the original stack trace of the crash to replicate.

We assess the performance of EvoCrash by conducting an empirical study on 54 crashes reported for real-world open-source Java projects. Our results show that EvoCrash can successfully replicate more crashes than STAR (+23%), MuCrash (+17%), and JCHARMING (+25%), which are the state-of-the-art tools based on crash stack traces. Furthermore, we observe that EvoCrash is not affected by the path explosion problem, which is a key problem for symbolic execution [81], and can mock environmental interactions which, in some cases, helps to cope with the environmental dependency problem.

Furthermore, we compare EvoCrash with EvoSuite to assess whether the crash replicated by our tools could be simply detected by classical coverage-based test case generators. The results of this comparison show that EvoCrash reproduced 85% of the crashes, while EvoSuite reproduced only 33% of them. For crashes reproduced by both EvoCrash and EvoSuite, on average, EvoCrash took 145 seconds while EvoSuite took 391 seconds. Thus, on average, EvoCrash is 170% more efficient than EvoSuite when they both reproduce crashes. These results show that coverage-based test generation lacks adequate guidance for crash reproduction. This in turn confirms the need for specialized search when the goal is to trigger specific software behavior rather than achieving high code coverage.

We also assess the extent of practical usefulness of the tests generated by EvoCrash during debugging and code fixing tasks. To this aim, we conducted a controlled experiment with 35 master students in computer science. The achieved results reveal that tests generated by EvoCrash increase participants' ability to provide fixes (+21% on average) while reducing the amount of time they spent to complete the assigned tasks (-15.36% on average).

The novel contributions of this extension are summarized as follows:

- A comparison of EvoCrash with EvoSuite, which is a test generation tool for coverage-based unit testing.
- A controlled experiment involving human participants; its results show that the usage of the tests aids developers in fixing the reported bugs while taking less time when debugging.
- We provide a publicly available replication package[1] that includes: (i) an executable jar of EvoCrash, (ii) all bug reports used in our study, (iii) the test cases generated by our tool, and (iv) anonymized experimental data as well as R scripts used to analyze the results from the controlled experiment.

The remainder of the chapter is structured as follows. Section 2.2 provides background on search-based software testing, in addition to describing the related work on the approaches to automated crash replication, unit test generation tools, and user studies in testing and debugging. Section 2.3 presents the EvoCrash approach. Section 2.4 and 2.5 describe the empirical evaluation of EvoCrash as well as the controlled experiment with human participants, respectively. Discussion follows in Section 2.6. Section 2.7 concludes the paper.

2.2 Background and Related Work

In this section, we present related work on automated crash reproduction, background knowledge on search-based software testing,related work in software testing and debugging which conducted experiments involving human participants.

2.2.1 Automated Approaches to Crash Replication

Previous approaches in the field of crash replication can be grouped into three main categories: (i) *record-replay* approaches, (ii) *post-failure* approaches using various

[1] **DOI**: 10.4121/uuid:001bb128-0a55-4a8d-b3f5-e39bfc5795ea

data sources, and (iii) *stack-trace based post-failure* techniques. The first category includes the earliest work in this field, such as ReCrash [58], ADDA [**?**], Bugnet [169], and jRapture [205]. In addition, [64] and [76] are recent record-replay techniques which are based on monitoring non-deterministic and hard-to-resolve methods (when using symbolic execution) respectively. The recent work on reproducing context-sensitive crashes of Android applications, MoTiF [114], also falls in the first category of record-replay techniques. The aforementioned techniques rely on program run-time data for automated crash replication. Thus, they record the program execution data in order to use it for identifying the program states and execution path that led to the program failure. However, monitoring program execution may lead to (i) substantial performance overhead due to software/hardware instrumentation [81,171,194], and (ii) privacy violations since the collected execution data may contain sensitive information [81].

On the other hand, post-failure approaches [137, 150, 151, 194, 217, 219] analyze software data (e.g., core dumps) only after crashes occur, thus not requiring any form of instrumentation. Rossler et al. [194] developed an evolutionary search-based approach named RECORE that leverages core dumps (taken at the time of a failure) to generate input data. RECORE combines the search-based input generation with a coverage-based technique to generate method sequences. Weeratunge et al. [208] used core dumps and directed search for replicating crashes related to concurrent programs in multi-core platforms. Leitner et al. [150, 151] used a failure-state extraction technique to create tests from core dumps (to derive input data) and stack traces (to derive method calls). Kifetew et al. [136, 137] used genetic programming requiring as input (i) a grammar describing the program input, and (ii) a (partial) call sequence. Boyapati et al. [69] developed another technique requiring manually written specifications containing method preconditions, postconditions, and class invariants. However, the above mentioned *post-failure* approaches need various types of information that are often not available to developers, thus decreasing their feasibility. To address lack of available execution data for replicating system-level concurrency crashes, Yu et al. [217] propose a new approach called, DESCRY. DESCRY only assumes the existence of the source code of processes under debugging and default logs generated by the failed execution. This approach [217] leverages a combination of static and dynamic analysis techniques and symbolic execution to synthesize the failure-inducing input data and interleaving schedule.

To increase the practical usefulness of automated approaches, researchers have focused on crash stack traces as the only source of information available for debugging. For instance, ESD [219] uses forward symbolic execution that leverages commonly reported elements in bug reports. BugRedux [134] also uses forward symbolic execu-

tion but it can analyze different types of execution data, such as crash stack traces. As highlighted by Chen and Kim [81], both ESD and BugRedux rely on forward symbolic execution, thus inheriting its problems due to *path explosion* and *object creation* [214]. As shown by Braione et al. [70], existing symbolic execution tools do not adequately address the synthesis of complex input data structures that require non-trivial method sequences. To address the path explosion and object creation problems, Chen and Kim [81] introduced STAR, a tool that applies backward symbolic execution to compute crash preconditions and generates a test using a method sequence composition approach. Despite these advances in STAR, Chen and Kim [81] reported that their approach is still affected by the path explosion problem when replicating some crashes. Therefore, path-explosion still remains an open issue for symbolic execution.

Different from STAR, JCHARMING [171, 172] uses a combination of crash traces and model checking to automatically reproduce bugs that caused field failure. To address the state explosion problem [60] in model checking, JCHARMING applies program slicing to direct the model checking process by reduction of the search space. Instead, MuCrash [215] uses mutation analysis as the underlying technique for crash replication. First, MuCrash selects the test cases that include the classes in the crash stack trace. Next, it applies predefined mutation operators on the tests to produce mutant tests that can reproduce the target crash.

STAR [81], JCHARMING [171,172], and MuCrash [215], have been empirically evaluated on a varying number of field crashes (52, 12, and 31, respectively) which were reported for different open source projects, including: Apache Commons Collections, Apache Ant, Apache Hadoop, Dnsjava, etc. The results of the evaluations are reported in the published papers, however, to the best of our knowledge, the tools are not publicly available.

A recent approach based on using crash stacks for reproducing concurrency failures, that violate thread safety of a class, is CONCRASH, proposed by Bianchi et al. [65]. As input, CONCRASH requires the class that violates thread safety and the generated crash stack trace. CONCRASH iteratively applies pruning strategies to search for test code and interleaving that trigger the target concurrency failure. Differently from our approach, CONCRASH targets only concurrency failures violating the thread-safety of a program [65], which represents the minority of failures reported in issue tracking systems [218]. For example, Yuan et al. [218] reported that only 10% of the failures in distributed data-intensive systems are due to multi-threaded inter-leavings. A later study by Coelho et al. [85] further reported that a large majority of failures in android apps are related to errors in programming logic and resource management, while concurrency accounts only for 2.9% of all failures.

In our earlier study [201], we investigated coverage-based unit testing tools like EvoSuite as a technology for replicating some crashes if relying on a proper fitness function specialized for crash replication. However, our preliminary results also indicated that this simple solution could not replicate some cases for two main reasons: (i) limitations of the developed fitness function, and (ii) the large search space in complex real-world software. The EvoCrash approach presented in this paper resumes this line of research because it uses evolutionary search to synthesize a crash reproducing test case. However, it is novel because it utilizes a more effective fitness function and it applies a Guided Genetic Algorithm (GGA) instead of coverage-oriented genetic algorithms. Section 2.3 presents full details regarding the novel fitness function and the GGA in EvoCrash.

2.2.2 Search-based Software Testing

Search-Based Software Testing (SBST) is a sub-field of a larger body of work on Search-Based Software Engineering (SBSE). In SBSE, software engineering tasks are reformulated as optimization problems, to which different meta-heuristic algorithms are applied to automate them [122]. As McMinn describes [161], search optimizations have been used in a plethora of software testing problems, including structural testing [209], temporal testing [187], functional testing [73], and mutation testing [132]. Among these, structural testing has received the most attention so far.

Applying an SBST technique on a testing problem requires [117,161]: (i) a representation for the candidate solutions in the search space, and (ii) a definition for a fitness function. The *representation* of the solutions shall constitute elements which make it possible to encode them using some data structures [122] (e.g., vectors, trees). This is mainly because search optimization techniques rely on operators that manipulate the encoded elements to derive new solutions. In addition, the representation shall be accurate enough so that a small change in one individual solution represents a neighbor solution in the search space [122].

A *fitness function* (also called objective or distance function) is used to measure the distance of each individual in the search space from the global optimum. Therefore, it is important that this definition is computationally inexpensive so that it could be used to measure the distance of multiple individuals until the global optimum is found [122].

Furthermore, as described before, path explosion and object creation are open problems when using symbolic execution [70] [81]. Different from symbolic execution, search-based software testing uses *distance functions* to satisfy each condition of the

program in "isolation" [59], i.e., independently from which alternative path is taken to reach the condition to solve. Focusing on each condition at a time allows to address the path explosion problem but, on the other hand, it may fail to capture dependencies between multiple conditions in the programs as in the case of *deceiving* conditions [160]. Search-based approaches can be implemented to handle complex input data type by relying on the APIs of the SUT. Indeed, random sampling is used to create randomized tests containing object references through the invocation of constructors and randomly generated method sequences. The "quality" of the generated test input data is then assessed through test execution and measuring the distance to satisfy a given branch. The complexity of the input is then evolved depending on whether more complex data structures help or not satisfying the testing criterion.

Moreover, with regards to environmental interactions, Arcuri et al. [54] show that such interactions may inhibit the success of automated test generation. This is mainly due to two reasons: (i) the code that depends on the environment may not be fully covered, and (ii) the generated tests may be unstable. Arcuri et al. [54] showed that proper instrumentation in a search-based test generator can be used not only to synthesize the test inputs during the search process but also to control the environmental state. More specifically, *mocking strategy* can be used to isolate the execution of a class from its environmental dependencies.

Finally, meta-heuristics that have been used in SBST include hill climbing, simulated annealing, genetic algorithms, and memetic algorithms. The first two algorithms fall in the category of *local* search techniques since they evaluate single points in the search space at the time [122]. On the other hand, genetic algorithms are *global* search techniques since they evaluate a population of candidate solutions from the search space in various iterations [122]. Memetic algorithms hybridize the local and global algorithms. Therefore, in these techniques, the individuals of populations in a global search are also provided with the opportunity for local improvements [106]. Since genetic algorithms have been widely applied to software testing problems, in what follows, we provide a brief description of a classic genetic algorithm.

2.2.2.1 Genetic Algorithms

Genetic Algorithms (GAs) imitate evolutionary processes observed in nature. A GA starts by initializing a random population of individuals. When applied to test generation problems, individuals are typically test suites comprised of test cases [103], or test cases consisting of a sequence of statements [178]. After the first population is initialized, tests are executed against the program under test and the best ones are selected to form new individuals. This process continues until either an individual that

satisfies the search criterion is found, or the allocated resources to the search process are consumed.

To produce the next generation, the best individuals from the previous generation (*parents*) are selected (*elitism*) and used to generate new test cases (*offspring*). Offspring is produced by applying typical evolutionary operators, namely crossover and mutation, to the selected "fittest" individuals. Depending on whether the parent or the offspring scores better for the search criterion, one is selected to be inserted into the next generation.

To illustrate the evolutionary operators, let us consider as examples two test cases $T_1 = \{s_1, \ldots, s_m\}$ and $T_2 = \left\{s^*_1, \ldots, s^*_n\right\}$ selected from a given generation as parents. To generate offspring O_1 and O_2, first a random number α, called *the relative cut-point*, between 0.0 and 1.0 is selected. Then, the first offspring O_1 will contain the first $\alpha \times m$ statements from T_1 followed by the last $(1 - \alpha) \times n$ statements from T_2. Similarly, O_2 will contain the first $\alpha \times n$ statements from T_2 followed by $(1 - \alpha) \times m$ statements from T_1. Thus, each offspring inherits its statements (e.g., objects instantiations, methods calls) from both the two parents.

Newly generated test cases are further changed by applying a mutation operator. With mutation, either random new statements are inserted into the tests, or random existing statements are removed, or random input parameters are modified [178]. Both crossover and mutation are performed such that the resulting test cases will be compilable. For example, if a new object is inserted as a parameter, then before it is inserted it is declared and instantiated.

2.2.3 Unit Test Generation Tools

A number of techniques and tools have been proposed in the literature to automatically generate tests maximizing specific code coverage criteria [18, 103, 112, 156, 159, 177, 184, 196, 206]. The main difference among them is represented by the core approach used for generating tests. For example, EvoSuite [103], JTExpert [196], and SAPIENZ [159] use genetic algorithms to create test suites optimizing code coverage; Randoop [177], T3 [184], Dynodroid [156], and Google Monkey [18] apply random testing, while DART [112] and Pex [206] are based on dynamic symbolic execution.

As reported in the related literature, such tools can be used to discover bugs affecting software code. Indeed, they can generate test triggering crashes when trying to generate tests exercising the uncovered parts of the code. For example, Fraser and Arcuri [101] successfully used EvoSuite to discover undeclared exceptions and bugs

in open-source projects. Recently, Moran et al. [165] used coverage-based tools to discover android application crashes. However, as also pointed out by Chen and Kim [81] coverage-based tools are not specifically defined for crash replication. In fact, these tools are aimed at covering all methods (and their code elements) in the class under test. Thus, already covered methods are not taken into account for search even if none of the already generated tests synthesizes the target crash. Therefore, the probability of generating tests satisfying desired crash triggering object states is particularly low for coverage-based tools [81].

On the other hand, for crash replication, not all methods should be exploited for generating a crash: we are interested in covering only a few lines in those methods involved in the failure, while other methods (or classes) might be useful only for instantiating the necessary objects (e.g., input parameters). Moreover, among all possible method sequences, we are interested only on those that can potentially lead to the target crash stack trace. Therefore, in this paper, we design and evaluate a tool-supported approach, named EvoCrash, which is specialized for stack trace based crash replication.

2.2.4 User Studies in Testing and Debugging

In 2005, Sjøberg et al. [200] conducted a survey in which they studied how controlled experiments are conducted in software engineering, in the decade from 1993 to 2002. As they report, 1.9% of the 5453 scientific articles reported controlled experiments in which human participants performed one or more software engineering tasks. Later on, in 2011, Buse et al. [74] surveyed over 3000 papers, spanning over ten years, to investigate trends, benefits, and barriers of involving human participants in software engineering research. As Buse et al. [74] report, about 10% of the surveyed papers involved humans to evaluate a research claim directly. As they observed, the number of papers in software engineering which use human evaluations is increasing, however, they highlighted that papers specifically related to software testing and debugging rarely involved human studies.

In the area of software testing, Orso and Rothermel [176] conducted a survey among 50 software testing scholars, to provide an account of the most successful research in software testing, since the year 2000. In addition, they aimed at identifying the most significant challenges and opportunities in the area. Orso and Rothermel [176] argue that while prominent advances have been made in empirical studies on software testing, more user studies, in particular within an industrial context, are needed in which practical impact of research becomes apparent. Ang et al. [44] recently stud-

ied the progress that is made in the research community since 2011 to address the suggestions given by Orso and Rothermel [176]. As their study indicates, involving human evaluations in studies on automated debugging techniques remains mostly unexplored.

Recently, some research work in software testing and debugging started involving user evaluations include the following: [181], [188], [79], [108], [191], [109], and [180]. Parnin and Orso [181] performed a preliminary study with 34 developers to investigate whether and to what extent using an automated debugging approach may aid developers in their debugging tasks. In their results, Parnin and Orso [181] show that several assumptions made by automated debugging techniques (e.g., examining isolated statements is enough to understand the bug and fix it) do not hold in practice. Moreover, Parnin and Orso [181] also encourage the researchers to involve developers in their studies to understand how richer information such as test cases and slices may make debugging aids more usable in practice.

Ramler et al. [188] compared tool-supported random test generation and manual testing, involving 48 master students. Their findings are twofold: (i) the number of detected defects by randomly generated test cases is in the range of manual testing, and (ii) randomly generated test cases detect different defects than manually-written unit tests.

Ceccato et al. [79] performed two controlled experiments with human participants to investigate the impact of using automatically generated test cases in debugging. They show that using automatically generated test cases has a positive impact on the accuracy and efficiency of developers working on fault localization and bug fixing tasks. Furthermore, Fraser et al. [108], and [109] conducted controlled experiments with human participants to investigate whether automatically generated unit test cases aid testers in code coverage and finding faults. In their experiments, they provided JavaDocs to the participants and asked them to both produce implementations and test suites. Their results confirmed that while automatically generated test cases, designed for high coverage, do not help testers find bugs, they do aid in achieving high coverage when compared to the ones produced by human participants.

In addition, Rojas et al. [191] combined a controlled experiment with 41 students with five think-aloud observations to assess the impact of using the automated test generation tool, *EvoSuite*, in software development. Their results confirmed that using the tool leads to an average branch coverage increase of 13%, and 36% less time spent on testing, compared to when developers write tests manually. The results from their think-aloud observations with professional programmers confirmed the necessity to (i) increase the usability of the tool, (ii) integrate it better during development, and

(iii) educate developers on how to best use the tool during development.

To improve the comprehensibility of test cases which in turn could improve the number of faults found by developers, Panichella et al. [180] proposed *TestDescriber* which automatically generates summaries of the portions of the code that is exercised by individual test cases. To assess the impact of their approach, Panichella et al. [180] performed a controlled experiment with 33 human participants comprising of professional developers, senior researchers, and students. The results of their study show that using TestDescriber, (i) developers find twice as many bugs, and (ii) test case summaries improve the comprehensibility of test cases which were considered useful by developers.

To investigate and understand the practical usefulness of automatically generated crash-reproducing tests, we acknowledge the need for involving human practitioners in our line of research. Therefore, as the first step in this direction, we conducted a controlled experiment (described in Section 2.5) with master students in computer science to assess the impact of using the crash-reproducing unit tests generated by EvoCrash when performing debugging tasks.

2.3 The EvoCrash Approach

In the following, we present the Guided Genetic Algorithm (GGA) and the fitness function we designed in our search-based approach to automated crash reproduction.

Figure 2.1 shows the main steps of EvoCrash. EvoCrash begins by pre-processing a crash stack trace log in order to formulate the target crash to be reproduced. Next, EvoCrash applies a Guided Genetic Algorithm (GGA) in order to search for a test case that triggers the same crash. The search is over either when the test is found or when the search budget is over. If a crash reproducing test case is found, it goes through post-processing, a phase where the generated test is minimized and transformed into an executable JUnit test. In what follows, we elaborate on each of the above phases in more detail.

2.3.1 Crash Stack Trace Processing

An optimal test case for crash reproduction has to crash at the same location as the original crash and produce a stack trace as similar to the original one as possible. Therefore, in EvoCrash we first parse the log file given as input in order to extract the crash stack frames of interest. A standard Java stack trace contains (i) the type of the

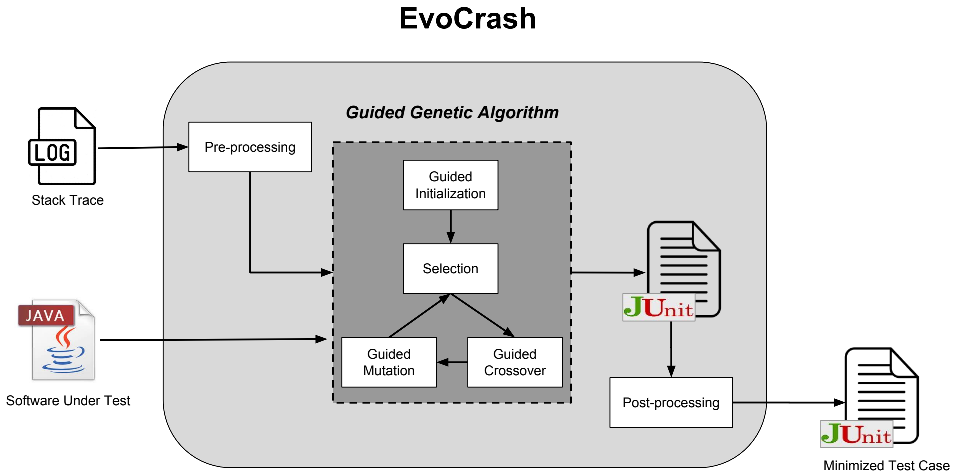


Figure 2.1: Overview of The Guided Genetic Algorithm in EvoCrash

exception thrown, and (ii) the list of stack frames generated at the time of the crash. Each stack frame corresponds to one method involved in the failure and contains: (i) the method name; (ii) the class name, and (iii) line numbers where the exception was generated. The last frame is where the exception has been thrown, whereas the root cause could be in any of the frames, or even outside the stack trace.

From a practical point of view, any class or method in the stack trace can be selected as code unit to use as input for existing test case generation tools, such as EvoSuite. However, since our goal is to synthesize a test case generating a stack trace as similar to the original trace as possible, we always target the class where the exception is thrown (last stack frame in the crash stack trace) as the main class under test (CUT).

2.3.2 Fitness Function

In search-based software testing, the fitness function is typically a *distance function* $d(.)$, which is equal to zero if and only if the a test case satisfying a given criterion is found. As described in our previous study [201], we have to consider three main conditions in the definition of our distance for crash replication: 1. the line (statement) where the exception is thrown has to be covered, 2. the target exception has to be thrown, and 3. the generated stack trace must be as similar to the original one as possible.

Therefore, we first define three different distance functions for the three conditions above, one for each condition. Then, we combine these three distances into our final fitness function using the sum-scalarization approach. The three distance functions as well as the final one are described in details in the following subsections.

Line distance. A test case t that successfully replicates a target crash has to cover the line of the production code where the exception was originally thrown. To guide the search toward covering the target line, we need to define a distance function $d_s(t)$ for line coverage. To this aim, we use two heuristics that have been successfully used in white-box testing for branch and statement coverage [160,201]: the *approach level* and the normalized *branch distance*. The *approach level* measures the distance in the control flow graph (i.e., the minimum number of control dependencies) between the path of the production code executed by t and the target line. The *branch distance* uses a set of well-established rules [160] to score how close t is to satisfy the conditional expression where the execution diverges from the paths to the target line.

Exception distance. The exception distance is used to check whether the test case t triggers the correct exception. Hence, we define the exception distance d_{except} as a boolean function that takes a zero value if and only if the target exception is thrown; otherwise, d_{except} is set to one.

Trace distance. Several stack trace similarity metrics have been defined in the related literature [90], although for different software engineering problems. These metrics could be in theory used to define our trace distance. Dang et al. [57, 90] proposed a stack trace similarity to clusterize duplicated bug reports. Their similarity metric uses dynamic programming to find the *longest common subsequence* (i.e., sequence of stack frames) among a pool of stack traces. The clusters are then obtained by applying a supervised hierarchical clustering algorithm [90]. However, this similarity metric requires a pool of stack traces plus a training algorithm to decide whether two stack traces are related to the same crash. Artzi et al. [57] proposed some similarity metrics to improve fault localization by leveraging concolic testing. Their intuition is that fault localization becomes more effective when generating passing test cases that are similar to the test cases inducing a failure [57]. However, the similarity metrics proposed by Artzi et al. cannot be used in our context for two main reasons: (i) the test inputs inducing the target failure are not available (generating tests that replicate a crash is the actual goal of EvoCrash and not its input) and (ii) the similarity metrics are defined for input and path-constraints (i.e., not for stack traces).

To calculate the trace distance, $d_{trace}(t)$, in our preliminary study [201] we used the distance function defined as follows. Let $S^* = \left\{e_1^*, \ \ldots, e_n^*\right\}$ be the target stack

trace to replicate, where $e_i^* = (C_1^*, m_1^*, l_1^*)$ is the i-the element in the trace composed by class name C_i^*, method name m_i^*, and line number l_i^*. Let $S = \{e_1, \ldots, e_k\}$ be the stack trace (if any) generated when executing the test t. The distance between the expected trace S^* and the actual trace S is defined as:

$$d_{trace}(t) = \sum_{i=1}^{\min\{k,n\}} \varphi\left(\text{diff}(e_i^*, e_i)\right) + |\, n - k \,| \tag{2.1}$$

where $\text{diff}(e_i^*, e_i)$ measures the distance between the two trace elements e_i^* and e_i in the traces S^* and S respectively; finally, $\varphi(x) \in [0, 1]$ is the widely used normalizing function $\varphi(x) = x/(x+1)$ [160]. However, such a distance definition has one critical limitation: it strictly requires that the expected trace S^* and the actual trace S share the same prefix, i.e., the first $\min\{k, n\}$ trace elements. For example, assume that the triggered stack trace S and target trace S^* have one stack trace element e_{shared} in common (i.e., one element with the same class name, method name, and source code line number) but that is located at two different positions, e.g., e_i^* is the second element in S ($e_{shared} = e_2$ in S) while it is the third one in S^* ($e_{shared} = e_3^*$ in S^*). In this scenario, Equation 2.1 will compare the element e_3^* in S^* with the element in S at the same position i (i.e., with e_3) instead of considering the closest element $e_{shared} = e_2$ for the comparison.

To overcome this critical limitation, in this paper we use the following new definition of stack trace distance:

Definition 1. *Let S^* be the expected trace, and let S be the actual stack trace triggered by a given test t. The stack trace distance between S^* and S is defined as:*

$$d_{trace}(t) = \sum_{i=1}^{n} \min\left\{\text{diff}(e_i^*, e_j) : e_j \in S\right\} \tag{2.2}$$

where $\text{diff}(e_i^*, e_j)$ measures the distance between the two trace elements e_i^* in S^* and its closest element e_j in S.

We say that two trace elements are equal if and only if they share the same trace components. Therefore, we define $\text{diff}(e_i^*, e_j)$ as follows:

$$\text{diff}(e_i^*, e_i) = \begin{cases} 3 & \text{if } C_i^* \neq C_i \\ 2 & C_i^* = C_i \text{ and } m_i^* \neq m_i \\ \varphi\left(|\, l_i^* - l_i \,|\right) \in [0;1] & \text{Otherwise} \end{cases} \tag{2.3}$$

The score $\text{diff}(e_i^*, e_i)$ is equal to zero if and only if the two trace elements e_i^* and e_i share the same class name, method name and line number. Similarly, $d_{trace}(t)$ in Equation 2.2 is zero if and only if the two traces S^* and S are equal, i.e., they share the same trace elements.

Table 2.1: Example of three different test cases with their corresponding distances and fitness function scores.

Test	d_s	d_{except}	d_{trace}	Fitness Function
t_1	0.14	1.00	2	$0.12 * w_1 + 1.00 * w_2 + 0.67 * w_3$
t_2	0.00	1.00	4	$0.00 * w_1 + 1.00 * w_2 + 4.00 * w_3$
t_3	0.00	0.00	5	$0.00 * w_1 + 0.00 * w_2 + 0.86 * w_3$

Final fitness function. To combine the three distances defined above, we use the *weighted-sum scalarization* [92].
Definition 2. *The fitness function value of a given test t is:*

$$f(t) = w_1 * \varphi(d_s(t)) + w_2 * d_{except}(t) + w_3 * \varphi(d_{trace}(t)) \quad (2.4)$$

where $d_s(t)$, $d_{except}(t)$, and $d_{trace}(t)$ are the three individual distance functions described above; $\varphi(.)$ is a normalizing function [160]; w_1, w_1, and w_3 are the linear combination coefficients.

Notice that in the equation above, the first and the last terms are first normalized before being summed up. This is because they have different orders of magnitude: the maximum value for $d_{trace}(t)$ corresponds to the total number of frames in the stack traces; $d_{except}(t)$ takes values in $\{0, 1\}$; while the maximum value of $d_s(t)$ is proportional to the cyclomatic complexity of the class under test.

In principle, the linear combination coefficients can be chosen such as to give higher priority to the different composing distances. In our context, meeting the three conditions for an optimal crash replication should happen in a certain order. In particular, executing the target line takes precedence over throwing the exception, and in turn, throwing the target exception takes priority over the degree to which the generated stack trace is similar to the reported one.

For example, let us consider the three test cases t_1, t_2, and t_3 reported in Table 2.1. In the example, t_1 does not cover the target line (i.e., $d_s(t_1) > 0$) and it throws an exception but not the target one; t_2 covers the target line but throws the wrong exception (i.e., $d_s(t_2) = 0$ and $d_{except} = 1.0$); finally, t_3 covers the target line (i.e., $d_s(t_2) = 0$), it throws the right exception (i.e., $d_{except} = 0$) but its trace similarity is larger than the one for t_1 (i.e., $d_{trace}(t_3) > d_{trace}(t_1)$). The distance values and the corresponding fitness function for the three tests are also reported in Table 2.1.

Now, let us suppose we decide to give larger priority to d_{trace} compared to the other distances, e.g., $w_1 = 0.05$, $w_2 = 0.05$, and $w_3 = 1$. By applying Equation 2, we

would obtain the following fitness scores:

$$f(t_1) = 0.05 * 0.12 + 0.05 * 1.00 + 0.67 \approx 0.7228$$
$$f(t_2) = 0.05 * 0.00 + 0.05 * 1.00 + 0.80 \approx 0.8500$$
$$f(t_3) = 0.05 * 0.00 + 0.05 * 0.00 + 0.86 \approx 0.8571$$

In other words, with these weights, t_3 has the largest (worst) fitness score although it is the closest one to replicate the target crash (it covers the target line and triggers the correct exception). Instead, t_1 and t_2 do not even cover the target line even though they have a better fitness than t_3. With the weights above, the corresponding fitness function $f(.)$ would misguide the search by introducing local optima. Therefore, our weights should satisfy the constraints $w_1 \geq w_3$ and $w_3 \geq w_1$, i.e., d_{trace} should not have larger a weight compared to the other distances.

Let us consider other three coefficients that satisfy the constraints above: $w_1 = 0.01$, $w_2 = 1$, $w_3 = 0.01$. The corresponding fitness values for the three tests in Table 2.1 are as follows:

$$f(t_1) = 0.01 * 0.12 + 1.00 + 0.01 * 0.67 \approx 1.0079$$
$$f(t_2) = 0.01 * 0.00 + 1.00 + 0.01 * 0.80 \approx 1.0080$$
$$f(t_3) = 0.01 * 0.00 + 0.00 + 0.01 * 0.86 \approx 0.0086$$

With these new weights, t_3 has the lowest (better) fitness value since both the two constraints $w_1 \geq w_3$ and $w_2 \geq w_3$ are satisfied. However, t_1 has a better fitness than t_2 although the latter covers the target line while the former does not. To avoid this scenario, our weights should satisfy another constraint: $w_1 \geq w_2 + w_3$.

In summary, choosing the weights for the function in Definition 2 consists in solving the following linear system of inequality:

$$\begin{cases} w_1 \geq w_2 + w_3 \\ w_1 \geq w_3 \\ w_2 \geq w_3 \end{cases} \tag{2.5}$$

In this paper, we chose as weights the smallest integer numbers that satisfy the two inequalities in the system above, i.e., $w_1 = 3$, $w_2 = 2$, $w_3 = 1$. With these weights, the fitness values for the test cases in the example of Table 2.1 become: $f(t_1) = 3.04$, $f(t_2) = 2.80$, and $f(t_3) = 0.86$. While choosing the smallest integers makes the interpretation of the fitness values simpler, we also used different integers in our preliminary trials. We did not observe any impact on the outcomes.

In general, with these weights, fitness function $f(t)$ assumes values within the interval $[0, 6]$; a value $3 \leq f(t) \leq 6$ indicates that a test t does not cover the target line; a

value $1 \leq f(t) < 3$ means that the test t covers the target line but does not throw the target exception; a zero value is reached if and only if the evaluated test t replicates the target crash.

2.3.3 Guided Genetic Algorithm

In EvoCrash, we use a novel genetic algorithm, named GGA (Guided Genetic Algorithm), suitably defined for the crash replication problem. While traditional search algorithms in coverage-based unit test tools target all methods in the CUT, GGA gives higher priority to those methods involved in the target failure. To accomplish this, GGA uses three novel *genetic operators* that create and evolve test cases that always exercise at least one method contained in the crash stack trace, increasing the overall probability of triggering the target crash. As shown in Algorithm 2.1 (*please see the end of the chapter*), GGA contains all main steps of a standard genetic algorithm: (i) it starts with creation of an initial population of random tests (line 5); (ii) it evolves such tests over subsequent generations using *crossover* and *mutation* (lines 12-20); and (iii) at each generation it *selects* the fittest tests according to the fitness function (lines 22-24). The main difference is represented by the fact that it uses (i) a novel routine for generating the initial population (line 5); (ii) a new crossover operator (line 15); (iii) a new mutation operator (lines 19-20). Finally, the fittest test obtained at the end of the search is post-processed (e.g., minimized) in line 26.

Initial Population. The routine used to generate the initial population plays a paramount role [179] since it performs sampling of the search space. In traditional coverage-based tools (e.g., EvoSuite [103] or JTExpert [196]) such a routine is designed to generate a well-distributed population (set of tests) that maximize the number of methods in the class under test C that are invoked/covered [103]. Instead, the main goal for crash replication is invoking the subset of methods M_{crash} in C that appear in the crash stack traces since they may trigger the target crash. Instead, the remaining methods can be still invoked with some random probability to instantiate objects (test inputs) or if they help to optimize the fitness function (i.e., decreasing the approach level and branch distance for the target line to cover).

For this reason, in this paper we use the novel routine highlighted in Algorithm 2.2 (*please see the end of the chapter*) for generating the initial sample for random tests. In particular, our routine gives higher importance to methods contained in crash stack frames. Subsequently, if a target call, selected by the developer, is public or protected, Algorithm 2.2 guarantees that this call is inserted in each test at least once. Otherwise, if the target call is private, the algorithm guarantees that each test contains at least

one call to a public caller method which invokes the target private call. Algorithm 2.2 generates random tests using the loop in lines 3-18, and requires as input (i) the set of public target method(s) M_{crash}, (ii) the population size N, and (iii) the class under test C. In each iteration, we create an empty test t (line 4) to fill with a random number of statements (lines 5-18). Then, statements are randomly inserted in t using the iterative routine in lines 8-18: at each iteration, we insert a call to one public method either taken from M_{crash}, or member classes of C. In the first iteration, crash methods in M_{crash} (methods of interest) are inserted in t with a low probability $p = 1/size$ (line 7), where $size$ is the total number of statements to add in t. In the subsequent iterations, such a probability is automatically increased when no methods from M_{crash} is inserted in t (line 15-17). Therefore, Algorithm 2.2 ensures that at least one method of the crash is inserted in each initial test[2].

The process of inserting a specific method call in a test t requires several additional operations [103]. For example, before inserting a method call m in t it is necessary to instantiate an object of the class containing m (e.g., calling one of the public constructors). Creating a proper method call also requires the generation of proper input parameters, such as other objects or primitive variables. For all these additional operations, Algorithm 2.2 uses the routine INSERT-METHOD-CALL (line 18). For each method call in t, such a routine sets each input parameter as follows:

Case 1 It re-uses an object or variables already defined in t with a probability p=1/3;

Case 2 If the input parameter is an object, it sets the parameter to `null` with a probability p=1/3;

Case 3 It randomly generates an objects or primitive value with a probability p=1/3;

Guided Crossover. Even if all tests in the initial population exercise one or more methods contained in the crash stack trace, during the evolution process—i.e., across different generations— tests can lose the inserted target calls. One possible cause for this scenario is the traditional *single-point* crossover, which generates two offsprings by randomly exchanging statements between two parent tests p_1 and p_2. Given a random cut-point μ, the first offspring o_1 inherits the first μ statements from parent p_1, followed by $| p_2 | - \mu$ statements from parent p_2. Vice versa, the second offspring o_2 will contain μ statements from parent p_2 and $| p_1 | - \mu$ statements from the parent p_1. Even if both parents exercise one or more failing methods from the crash stack trace, after crossover is performed, the calls may be moved into one offspring only. Therefore, the traditional single-point crossover can hamper the overall algorithm.

[2]In the worst case, a failing method will be inserted at position $size$ in t since the probability $insert_probability$ will be $1/(size - size + 1) = 1$

To avoid this scenario, GGA leverages a novel *guided single-point crossover* operator, whose main steps are highlighted in Algorithm 2.3 (*please see the end of the chapter*). The first steps in this crossover are identical to the standard single-point crossover: (i) it selects a random cut point μ (line 5), (ii) it recombines statements from the two parents around the cut-point (lines 7-8 and 12-13 of Algorithm 2.3). After this recombination, if o_1 (or o_2) loses the target method calls (a call to one of the methods reported in the crash stack trace), we reverse the changes and re-define o_1 (or o_2) as pure copy of its parent p_1 (p_2 for offspring o_2) (`if` conditions in lines 10-11 and 16-17). In this case, the mutation operator will be in charge of applying changes to o_1 (or o_2).

Moving method calls from one test to another may result in non-well-formed tests. For example, an offspring may not contain proper class constructors before calling some methods; or some input parameters (either primitive variables or objects) are not inherited from the original parent. For this reason, Algorithm 2.3 applies a *correction* procedure (lines 9 and 15) that inserts all required objects and primitive variables into non-well-formed offspring.

Guided Mutation. After crossover, new tests are usually mutated (with a low probability) by adding, changing and removing some statements. While adding statements will not affect the type of method calls contained in a test, the statement deletion/change procedures may remove relevant calls to methods in the crash stack frame. Therefore, GGA also uses a new *guided-mutation* operator, described in Algorithm 2.4 (*please see the end of the chapter*).

Let $t = \langle s_1, \ldots, s_n \rangle$ be a test case to mutate, the *guided-mutation* iterates over the test t and mutates each statement with probability $1/n$ (main loop in lines 4-15). Inserting statements consists of adding a new method call at a random point $i \in [1; n]$ in the current test t (lines 12-13 in Algorithm 2.4). This procedure also requires to instantiate objects or declare/initialize primitive variables (e.g., integers) that will be used as input parameters.

When changing a statement at position i (in lines 10-11), the mutation operator has to handle two different cases:

Case 1 if the statement s_i is the declaration of a primitive variable (e.g., an integer), then its primitive value is changed with another random value (e.g., another random integer);

Case 2 if s_i contains a method or a constructor call m, then the mutation is applied by replacing m with another public method/constructor having the same return type; the input parameters (objects or primitive values) are either (i) taken from

the previous $i - 1$ statements in t, (ii) set to `null` (for objects only), (iii) or randomly generated. These three mutations are applied with the probability $p = 1/3$. Therefore, they are equally probable and mutually exclusive for each input parameter.

Finally, removing a method call (lines 8-9 in Algorithm 2.4) also requires to delete the corresponding variables and objects used as input parameters (if such variables and objects are not used by any other method call in t). If the test t loses the target method calls (i.e., methods in M_{crash}) because of the mutation, then the loop in lines 4-15 is repeated until one or more target method calls are re-inserted in t; otherwise the mutation process terminates.

Post processing. At the end of the search process, GGA returns the fittest test case according to our fitness function. The resulting test t_{best} can be directly used by a developer as a starting point for crash replication and debugging.

Since method calls are randomly inserted/changed during the search process, the final test t_{best} can contain statements not needed to replicate the crash. For this reason, GGA post-processes t_{best} to make it more concise and understandable. For this post-processing, we reused the *test optimization* routines available in EvoSuite [103], namely: *test minimization*, and *values minimization*. *Test minimization* applies a simple greedy algorithm: it iteratively removes all statements that do not affect the final fitness value. Finally, randomly generated input values can be hard to interpret for developers [41]. Therefore, the *values minimization* from EvoSuite shortens the identified numbers and simplifies the randomly generated strings [102].

2.3.4 Mocking Strategies

Since EvoCrash is built on top of EvoSuite, by default, EvoCrash inherits the mocking strategies implemented in EvoSuite [53–55]. Therefore, if reproducing a target crash requires environmental interactions involving system calls (e.g., System.currentTimeMillis), network connections (e.g., calls to java.net APIs) and file system (e.g., calls to java.io.File), EvoCrash benefits from the available mocking operators to reproduce the crash.

However, it is possible that reproducing a crash requires specific content as the result of the interaction with the environment. For example, it could be that specific content of an XML file is needed to reproduce a crash. In these cases, EvoCrash lacks support for finding the specific content needed to optimize the fitness function. This is an

open problem in automated test generation that calls for future work and is beyond the scope of this study.

2.4 Study I: Effectiveness

This section describes the empirical study we conducted to benchmark the effectiveness of the EvoCrash approach.

2.4.1 Research Questions

To evaluate the effectiveness of EvoCrash we formulate the following research questions:

- **RQ1:** *How does EvoCrash perform compared to coverage-based test generation?* EvoCrash is built on top of Evosuite, which is a coverage-based test generation tool for unit testing. Therefore, with this research question, we aim at investigating to what extent EvoCrash actually provides the expected benefits in terms of the number of reproduced crashes and test generation time compared to a classical coverage-based test generation approach.
- **RQ2:** *In which cases can EvoCrash successfully reproduce the targeted crashes, and under what circumstances does it fail to do so?* With this research question, we aim at evaluating the capability of our tool to generate test cases (i) that can replicate the target crashes, and (ii) that are useful for debugging.
- **RQ3:** *How does EvoCrash perform compared to state-of-the-art reproduction approaches based on stack traces?* In this research question, we investigate the advantages and disadvantages of EvoCrash as compared to the most recent stack trace based approaches to crash reproduction previously proposed in the literature.

For **RQ1**, we selected EvoSuite [103] as a representative tool for state-of-the-art approaches for coverage-based unit testing. Our choice is guided by the fact that EvoSuite won the latest two editions of the SBST tool competition [115] [107] and achieved very competitive scores (i.e., code coverage and fault detection rate) compared to hand-written tests. Moreover, EvoCrash and EvoSuite share the same instrumentation engine, the test execution environment and the encoding schema for test cases. By default, EvoSuite uses the *Archive-based Whole Test Suite* generation approach (WSA) [193], which evolves test suites and optimizes multiple testing criteria.

The default coverage criteria are *line coverage, branch coverage, direct branch coverage, weak mutation, exception coverage, no-exception top-level method coverage, and output coverage*, which are described in detail by Rojas et al. [190]. Exception coverage is particularly important in our context: using WSA, when this criterion is enabled, EvoSuite stores in an *archive* all test cases (which compose candidate test suites) that trigger an exception when trying to maximize the other coverage criteria. Therefore, the final test suite produced from EvoSuite not only achieves higher code coverage but also contains all tests triggering some exceptions which were found during the generation process.

For the sake of our analysis, we conducted the experiments with EvoSuite using the default coverage criteria and targeting the same class tested by EvoCrash. First, we compare EvoSuite and EvoCrash in terms of crash replication frequency, i.e., the number of times each of the two techniques successfully reproduced a crash over 15 independent runs. A crash is covered, according to the *Crash Coverage* criterion by Chen and Kim [81], when the test generated by one tool triggers the same type of exception at the same crash line as reported in the crash stack trace. Therefore, for this criterion, we classified as *covered* only those crashes for which EvoCrash reached a zero-fitness value, i.e., when the generated crash stack trace is identical to the target one.

While EvoCrash produces only one test for each crash, EvoSuite generates entire test suites. Thus, for the latter tool, we consider a crash as replicated if at least one test case within the test suite generated by EvoSuite is able to replicate the target crash. To further guarantee the reliability of our results, we re-executed the tests generated by EvoCrash and EvoSuite against the CUT to ensure that the crash stack frame was correctly replicated.

We also compared EvoSuite and EvoCrash in terms of search time required to replicate each crash. To this aim, during each tool run, we stored the duration of the time interval between the start of the search and the point in time where each test case (or test suite for EvoSuite) was generated. Then, the time to replicate each crash (if replicated) corresponds to the search time interval of the test case (or test suite) that successfully replicates it.

To address **RQ**$_2$, we apply the two criteria proposed by Chen and Kim [81] for evaluating the effectiveness of crash replication tools: *Crash Coverage* and *Test Case Usefulness*. *Crash Coverage* is the same criterion used to answer **RQ**$_1$. For the *Test Case Usefulness*, a test case generated by EvoCrash is considered *useful* if and only if it reveals the actual bug that causes the original crash. According to the guidelines in [81], a test case *reveals a bug* if the generated crash trace includes the buggy frame (i.e., the stack element which the buggy method lies in [81]) or the frame the execution

of which covers the buggy statement. The guidelines in [81] further clarify that in addition to generating the buggy frame, *useful* tests have to reveal the origin of the corrupted input values (e.g., `null` values) passed to the buggy methods that trigger the crash [81]. This implies that if the buggy frame receives input arguments, then a *useful* test case must also generate at least one frame at a higher level than the buggy frame, through which we can observe how the input arguments to the buggy method are generated. Of course, if a) the stack trace has only one frame, or 2) the buggy method does not receive any arguments, then a *useful* test must only generate the buggy frame to be considered as useful.

To assess usefulness of the tests, we carefully inspected the original developers' fixes to identify the bug fixing locations. We manually examined each crash classified as *covered* (using the coverage criterion) to investigate if it reveals the actual bug following the guidelines in [81]. This manual validation has been performed by the first two authors independently, and cases of disagreement were discussed and resolved.

For **RQ3**, we selected three state-of-the-art techniques, namely: STAR [81], MuCrash [215], and JCHARMING [171, 172]. These three techniques are modern approaches to crash replication for Java programs, and they are based on three different categories of algorithms: symbolic execution [81], mutation analysis [215], and model checking [171].

At the time of writing this paper, STAR, MuCrash, and JCHARMING were not available (either as executable jars or source code). Therefore, to compare our approach, we rely on their published data. Thus, we compared EvoCrash with MuCrash for the 12 bugs selected that have also been used by Xuan et al. [215] to evaluate MuCrash. We compared EvoCrash with JCHARMING for the 13 bug reports that have been also used by Nayrolles et al. [171]. Finally, we compare EvoCrash with STAR for the 51 bugs in our sample that are in common with the study by Chen and Kim [81].

2.4.2 Definition and Context

As Table 2.2 presents, the *context* of this study consists of 54 bugs from seven real-world open source projects: `Apache Commons Collections`[3] (ACC), `Apache Ant`[4] (ANT), `Apache Log4j`[5] (LOG), `ActiveMQ`[6], `DnsJava`[7], and `JFreeChart`[8].

[3] https://commons.apache.org/proper/commons-collections/
[4] http://ant.apache.org
[5] http://logging.apache.org/log4j/2.x/
[6] http://activemq.apache.org/
[7] http://www.dnsjava.org/
[8] http://jfree.org/jfreechart//

ACC is a popular Java library with 25,000 lines of code (LOC), which provides utilities to extend the Java Collection Framework. For this library, we selected 12 bug reports publicly available on `Jira`[9] submitted between October 2003 and June 2012 and involving five different ACC versions.

ANT is a large Java build tool with more than 100,000 LOC, which supports different built-in tasks, including compiling, running and executing tests for Java applications. For ANT we selected 21 bug reports submitted on `Bugzilla`[10] between April 2004 and August 2012 and that concern 10 different versions and sub-modules.

LOG is a widely-used Java library with 20,000 LOC that implements logging utilities for Java applications. For this library we selected 18 bug reports reported within the time window between June 2001 and October 2009 and that are related to three different LOG versions.

ActiveMQ is a messaging and Integration Patterns server that is actively maintained by the Apache Software Foundation. ActiveMQ has 205000 LOC, and supports many cross-language clients written in Java, C, C++, C#, and PHP. We selected one case from ActiveMQ that was also used for evaluating JCHARMING.

DnsJava is an implementation of DNS in Java, which has more than 3000 LOC. It supports all defined record types (including the *DNSSEC* types), and unknown types. It can be used for queries, zone transfers, and dynamic updates. It includes a cache which can be used by clients, and a minimal implementation of a server. In addition, since it is written in pure Java, DnsJava is fully threadable. We selected one case from DnsJava, which was also used in the evaluation of JCHARMING [171, 172].

JFreeChart is a free Java chart library, with 310000 LOC, that could be used to display high-quality charts in both server-side and client-side applications. JFreeChart has a well-documented API and it has been maintained over a long period of time, since 2005. We also selected a case from JFreeChart to use for comparison with JCHARMING.

We selected this set of bugs as they have been used in the previous studies on automatic crash reproduction when evaluating symbolic execution [81], mutation analysis [215], and directed model checking [171] and other tools [82, 129]. The characteristics of the selected bugs, including type of exception and priority, are summarized in Table 2.2. These bugs cover crashes that involve the most common Java `Exceptions` [84], such as `NullPointerException` (74%), `ArrayIndexOutOf-`

[9] `https://issues.apache.org/jira/secure/Dashboard.jspa`
[10] `https://bz.apache.org/bugzilla/`

Table 2.2: The 54 real-world bugs used in our study.

Project	Bug IDs	Versions	Exceptions	Priority	Ref.
ACC	4, 28, 35 48, 53, 68 70, 77, 104 331, 277, 411	2.0 - 4.0	NullPointer (5) UnsupportedOperation (1) IndexOutOfBounds (1) IllegalArgument (1) ArrayIndexOutOfBounds (2) ConcurrentModification (1) IllegalState (1)	Major (10) Minor (2)	[81] [215]
ANT	28820, 33446, 34722 34734, 36733, 38458 38622, 41422, 42179 43292, 44689, 44790 46747, 47306, 48715 49137, 49755, 49803 50894, 51035, 53626	1.6.1 - 1.8.2	ArrayIndexOutOfBounds (3) NullPointer (17) StringIndexOutOfBounds (1)	Critical (2) Major (5) Medium (14)	[81] [172]
LOG	29, 43, 509, 10528 10706, 11570, 31003 40212, 41186, 44032 44899, 45335, 46144 46271, 46404, 47547 47912, 47957	1.0.2 - 1.2	NullPointer (17) InInitializerError (1)	Critical (1) Major (4) Medium (11) Enhanc. (1) Blocker (1)	[81] [172]
ActiveMQ	5035	5.9	ClassCastExecption (1)	Major (1)	[172]
DnsJava	38	2.1	ClassCastException (1)	N/A (1)	[172]
JFreeChart	434	1.0	NullPointerException (1)	N/A (1)	[172]

`BoundsException` (9%), `IllegalStateExcep-tion` and `IllegalArgumentE-xception` (3%). Furthermore, the severity of these real-world bugs varies between *medium* (46%), *major* (37%) and *critical* (5%) as judged by the original developers.

50 of these cases come from the primary study we performed in [203]. In this extension to [203], we aimed at extending the comparison with JCHARMING via the cases reported in [172]. However, ultimately, we chose to discard several cases reported in [172], and extend the comparison with JCHARMING via only 4 new cases, for four main reasons:

1. In six cases, the exact buggy version of the target software was either unknown or not found. Consequently, the reported line numbers in stack traces did not match the source code. Since the fitness function (Section 2.3.2) is primarily designed based on the exact line numbers where the exceptions are thrown, we discarded such cases.

2. As Nayrolles et al. report [172], to make a trade-off between reproducibility and relevance of the test cases, after a number of incremental attempts, they arrived at the threshold of 80% for reproducing stack traces. Thus, in some cases they report *partial* coverage, which means that at least 80% of a stack trace could be reproduced in those cases. While this partial measure is relative to the size of the stack traces, in our case we need to have exact measure of the reproduced traces to compare the *usefulness* of the tests, as described in Section 2.4.1.

3. In two cases, `ActiveMQ-1054` and `ArgoUML-311`, the reported stack traces lack line numbers. Thus, considering how the fitness function works (Section 2.3.2), we could not apply EvoCrash on such cases.

4. Finally, one of the reported cases in [172], `Mahout-1594`, actually refers to an external problem in the configuration file. Thus, this case was not a valid crash case to be considered in this study.

2.4.3 Experimental Procedure

We run EvoCrash and EvoSuite on each target crash to try to generate a test case and test suite able to reproduce the corresponding stack trace. Given the randomized nature of genetic algorithms, we executed the tools multiple times to verify that the target crashes are replicated in most of the runs. For RQ_1, we ran EvoSuite and EvoCrash 15 times for each crash. For RQ_2 the search for each target bug/crash was repeated 50 times.

In our experiment, we configured both tools by using standard parameter values widely used in evolutionary testing [52, 103, 178]:

- **Population size**: we use a population size of 50 individuals as suggested in [103, 178]. In the context of EvoCrash, individuals are test cases whereas in the context of EvoSuite, individuals are test suites, containing one or more test cases.

- **Crossover**: For EvoCrash, we use the novel *guided single-point* crossover; in EvoSuite, the crossover operator is the classic *single-point* crossover [103]. In both cases, the *crossover probability* is set to $p_c 0.75$ [103].

- **Mutation**: EvoCrash uses our *guided uniform mutation*, which mutates test cases by randomly adding, deleting, or changing statements. EvoSuite uses the standard *uniform mutation*, which randomly adds, deletes, or changes test cases in a test suite. For both cases, we set the *mutation probability* equal to $p_m 1/n$, where n is the length of the test case/suite taken as input [103].

- **Search Timeout**: The choice of 10 minutes as the search budget is a common practice in studies on search-based test generation [52, 103, 178]:. In our preliminary experiments, we noticed that the number of reproduced crashes does not change after 10 minutes. Therefore, in both cases, the search stops when a zero-fitness function value is detected or when the timeout of 10 minutes is reached.

2.4.4 Comparison with Coverage-Based Test Generation

As Table 2.3 shows, EvoCrash reproduced 46 crashes (85%) out of 54, compared to 18 crashes (33%) that were reproduced by EvoSuite. In particular, 28 (52%) crashes out of 54 were reproduced only by EvoCrash. Other 18 crashes (33%) were reproduced by both EvoCrash and EvoSuite. Finally, for the remaining 8 cases (14%) both EvoCrash and EvoSuite failed to generate a crash reproducing test.

However, in those 18 cases where both EvoSuite and EvoCrash generate tests, the former always achieved a lower or equal reproduction rate compared to the latter, i.e., every crash was rarely reproduced out of 15 runs (e.g., ACC-53 in Table 2.3). Furthermore, EvoSuite took longer compared to EvoCrash to reproduce the same crashes. Indeed, EvoCrash took 145 seconds on average to reproduce the crashes, while EvoSuite required 391 seconds (+170%) to reproduce the same crashes on average.

Table 2.3: Crash reproduction results for comparing Archive-based Whole Test Suite generation (WSA) in EvoSuite and Guided Genetic Algorithm (GGA) in EvoCrash. The bold cases are the ones for which only EvoCrash could generate a test at least 8 times out of 15 runs.

		EvoCrash		**EvoSuite**	
Project	**Bug ID**	**avg. time**	**reproduction %**	**avg. time**	**reproduction %**
ACC	4	2	100%	314	100%
	28	1	100%	10	100%
	35	1	100%	50	100%
	48	40	100%	350	33%
	53	5	100%	377	66%
	68	600	0%	600	0%
	70	2	100%	407	33%
	77	98	100%	233	100%
	104	455	73%	600	0%
	331	100	73%	315	20%
	377	100	100%	335	13%
	411	153	80%	600	0%
	28820	600	0%	600	0%
	33446	10	100%	540	40%
	34722	59	100%	459	26%
	34734	45	100%	600	0%
	36733	32	86%	600	0%

	38458	43	86%	510	20%
	38622	33	100%	81	100%
	41422	220	66%	590	13%
	42179	56	93%	77	60%
	43292	600	0%	600	0%
	44689	32	100%	358	40%
	44790	15	100%	540	40%
	46747	600	0%	600	0%
	47306	600	0%	600	0%
	48715	600	0%	600	0%
	49137	90	100%	320	100%
	49755	30	100%	449	53%
	49803	10	100%	600	0%
	50894	42	100%	600	0%
	51035	600	0%	600	0%
	53626	105	100%	600	0%
	29	28	93%	301	6%
	43	600	0%	600	0%
	509	136	100%	600	0%
	10528	1	100%	3	100%
	10706	1	100%	35	100%
	11570	1	100%	129	100%
	31003	1	100%	9	93%
	40212	18	100%	472	26%
	41186	1	100%	27	100%
LOG	**44032**	3	100%	487	33%
	44899	42	100%	69	93%
	45335	10	100%	462	46%
	46144	20	93%	533	13%
	46271	3	100%	74	100%
	46404	59	100%	600	0%
	47547	3	100%	10	100%
	47912	38	93%	388	40%
	47957	5	100%	28	100%
ActiveMQ	**5035**	377	60%	600	0%
DnsJava	**38**	115	85%	481	13%
JFreeChart	**434**	389	53%	500	13%

The results above show that indeed the GGA in EvoCrash outperforms WSA in EvoSuite for crash reproduction in both the number of reproduced crashes and test generation times. The underlying explanation for such observations is that EvoSuite, using WSA, evolves test suites with the goal of maximizing code coverage. Assuming that line l is where the target exception e happens, if there is a test suite that includes a test case t_l that covers l, EvoSuite archives t_l and l, and proceeds by evolving test suites targeting only the remaining uncovered lines. The archived test case t_l that covers the target line l, by chance may or may not trigger e as well. Furthermore, since criterion *Exception* was included in the optimization criteria, if there exists a test suite that contains test case t_e which triggers an exception, EvoSuite would archive t_e. By chance, t_e may or may not trigger e on the target line l.

On the other hand, EvoCrash uses GGA, which customizes test generation for crash coverage. Therefore, the search is aimed for a test case that both covers the target line l, and triggers the target exception e. This means that even if a test t_i covers l, EvoCrash keeps t_i in the search process in order to evolve it until it can also trigger e.

Thus, this comparison highlights that while coverage-based test generation by EvoSuite may by chance detect crashes, using GGA is a more effective and efficient approach for crash reproduction.

2.4.5 Crash Reproduction Effectiveness

This section presents the results of the empirical study we conducted to evaluate the effectiveness of EvoCrash in terms of crash coverage and test case usefulness.

Table 2.4: Detailed crash reproduction results, where **Y(Yes)**, indicates the capability to generate a useful test case, **N(No)** indicates lack of ability to reproduce a crash, **NU(Not Useful)** shows that a test case could be generated, but it was not useful, and **'-'** indicates that data regarding the capability of the approach in reproducing the identified crash is missing. The bold cases are the ones for which only EvoCrash could generated a test and the underlined ones are those where EvoCrash failed to produce a test at least 25 times out of 50 runs.

Project	Bug ID	EvoCrash	STAR [81]	MuCrash [215]	JCHARMING [171]
	4	Y	Y	Y	-
	28	Y	Y	Y	-
	35	Y	Y	Y	-
	48	Y	Y	Y	-
	53	**Y**	Y	**N**	-
ACC					

	68	N	N	N	-
	70	**Y**	**N**	**N**	-
	77	NU	NU	N	-
	104	N	Y	Y	-
	331	**Y**	**N**	Y	-
	377	**Y**	**N**	Y	-
	411	Y	Y	Y	-
	28820	N	N	-	-
	33446	NU	NU	-	-
	34722	**Y**	**N**	-	-
	34734	**NU**	**N**	-	-
	36733	NU	NU	-	-
	38458	Y	Y	-	-
	38622	Y	Y	-	Y
	41422	**NU**	Y	-	**N**
	42179	**Y**	**N**	-	-
	43292	N	Y	-	-
ANT	**44689**	**Y**	**NU**	-	-
	44790	Y	Y	-	-
	46747	N	N	-	-
	47306	N	N	-	-
	48715	N	N	-	-
	49137	**Y**	**NU**	-	-
	49755	Y	Y	-	-
	49803	Y	Y	-	-
	50894	**Y**	**NU**	-	-
	51035	N	N	-	-
	53626	**Y**	**N**	-	-
	29	Y	Y	-	-
	43	N	N	-	-
	509	**Y**	**N**	-	-
	10528	**Y**	**N**	-	-
	10706	**Y**	**N**	-	-
	11570	Y	Y	-	Y
	31003	Y	Y	-	-
	40212	**Y**	**NU**	-	Y
	41186	Y	Y	-	Partial
LOG	**44032**	**Y**	**N**	-	-
	44899	**Y**	**N**	-	-

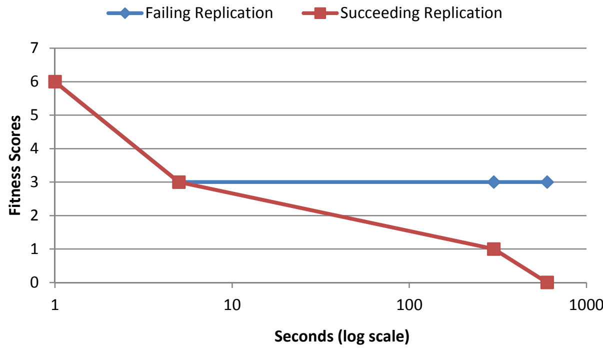


Figure 2.2: Fitness progress over time for both succeeding and failing runs of EvoCrash for ACC-104.

	45335	**Y**	**NU**	-	**N**
	46144	**Y**	**N**	-	-
	46271	NU	Y	-	Y
	46404	**Y**	**N**	-	-
	47547	Y	Y	-	-
	47912	**Y**	**NU**	-	Y
	47957	**NU**	Y	-	**N**
ActiveMQ	**5035**	**Y**	-	-	**N**
DnsJava	38	Y	-	-	Y
JFreeChart	434	Y	-	-	Y

EvoCrash Results (RQ2) As Table 2.4 illustrates, EvoCrash can successfully replicate the majority of the crashes in our dataset. 39 cases could be replicated 50 times out of 50 runs of EvoCrash. Of the replicated cases, LOG-509 had the lowest rate of replications - 39 out of 50. EvoCrash reproduces 11 crashes out of 12 (91%) for ACC, 15 out of 21 (71%) for ANT, and 17 out of 18 (94%) for LOG. Overall, EvoCrash can replicate 46 (85%) out of the 54 crashes.

To assess the usefulness of the generated test cases, as explained in Sub-section 2.4.1, we used the same criterion that was used for STAR [81]. Based on this, 38 (84%) of the replications were useful, as they included the buggy frame. The remaining 16% non-useful replications were mainly due to having dependency on data from external files which were not available during replication.

For ACC, ACC-68 was not reproducible by EvoCrash. In this case, the class under test includes three nested classes, and the inner-most one was where the crash occurs. We could not replicate this crash as EvoCrash relies on the instrumentation engine of EvoSuite, which does not currently support the instrumentation of multiple inner classes.

In addition, for ACC-104 [11], EvoCrash could replicate the case 42 times out of 50. The average time EvoCrash took for reproducing this case is 300 seconds. In this case, the defect lies on line *20* in Figure 2.1, where the shift operation does not correctly increment or decrement array indexes. In order to replicate this case, a test case shall meet the following criteria: 1) Make an object of the `BoundedFifoBuffer` class. 2) Add an arbitrary number of objects to the buffer. 3) Remove the last item from the buffer, and add arbitrary number of new items. 4) Remove an item that is not the last item in the buffer.

To understand why EvoCrash takes relatively longer to reproduce ACC-104, Figure 2.2 demonstrates the search progress during the failing and successful executions. As the Figure shows, during the failing executions, the fitness value quickly progresses to *3.0* and it remains unchanged until the search budget *(10 minutes)* is over. In these executions, a fitness value of *3.0* means that the target line, line *20* in Figure 2.1 is covered by the execution of the test cases. However, the target exception `ArrayIndexOutOfBounds` is not thrown at this line, which is why the fitness does not improve and remains *3.0* until the search time is consumed. On the other hand, during the successful runs, not only line *20* is covered, on average in five seconds, but also after 5 minutes, the target exception is thrown and generates the reported crash stack trace. As our results indicate, setting an object of `BoundedFifoBuffer` to the right state such that an arbitrary number of elements are added and removed in a certain order (as indicated previously) to throw the `ArrayIndexOutOfBounds` exception is a challenging task.

For ANT, six of the 20 crashes (30%) are currently not supported by EvoCrash. For these cases, the major hindering factor was the dependency on a missing external `build.xml` file, which is used by ANT for setting up the project configurations. However, `build.xml` was not supplied for many of the crash reports. In addition, the use of Java reflection made it more challenging to reproduce these ANT cases, since the specific values for class and method names are not known from the crash stack trace. For LOG, one of the 18 cases (5%) is not supported by EvoCrash. In this case, the target call is made to a static class initializer, which is not supported by EvoCrash yet.

[11] https://issues.apache.org/jira/browse/COLLECTIONS-104

```
java.lang.ArrayIndexOutOfBoundsException:
    at org.apache.commons.collections.buffer.BoundedFifoBuffer.remove(
    BoundedFifoBuffer.java:347)
```

Listing 2.2: Crash Stack Trace for ACC-104.

```
public void remove() {
  if (lastReturnedIndex == -1) {
     throw new IllegalStateException();
  }

  // First element can be removed quickly
  if (lastReturnedIndex == start) {
     BoundedFifoBuffer.this.remove();
     lastReturnedIndex = -1;
     return;
  }

  // Other elements require us to shift the
      subsequent elements
  int i = lastReturnedIndex + 1;
  while (i != end) {
     if (i >= maxElements) {
        elements[i - 1] = elements[0];
        i = 0;
     } else {
        elements[i - 1] = elements[i];
        i++;
     }
  }

  lastReturnedIndex = -1;
  end = decrement(end);
  elements[end] = null;
  full = false;
  index = decrement(index);
}
```

Listing 2.1: Buggy method for ACC-104.

2.4.6 Comparison to State of the Art

This section discusses the results of the comparison between EvoCrash and the state-of-the-art approaches based on crash stack traces, namely STAR [81], MuCrash [215], and JCHARMING [171]. In Table 2.4, bold entries represent bugs which can be

triggered by EvoCrash, but not by at least one of the other techniques; Underlined entries represent bugs that EvoCrash cannot reproduce, while there is another technique that can. As can be seen, there are 23 (bold) cases in which EvoCrash outperforms the state of the art, and there are two (underlined) cases that EvoCrash cannot handle. Below we discuss these cases in more detail.

EvoCrash *vs.* STAR. As Table 2.4 presents, for ACC, EvoCrash covers all the cases that STAR covers. In addition, EvoCrash covers three cases (25%) which were not covered by STAR due to the path explosion problem. For instance, in ACC-331, the defect exists in a private method, `least`, inside a for loop, inside the third `if` condition, which could not be handled by STAR.

For ANT, EvoCrash supports seven cases (35%) which are not covered by STAR. Out of the seven, there are three cases, for which only EvoCrash can generate a useful test case. Listing 2.3 shows the crash stack trace for one of these cases (ANT-49137). As reported in the issue tracking system of the project[12], in this case, the defect exists in the 4th stack frame. Thus, a useful test case should (i) make a call to the method `delete`, (ii) trigger a `java.lang.NullPointerException`, and (iii) yield a crash trace which includes the first stack frame, which is where the exception was thrown.

```
1 java.lang.NullPointerException:
2   at org.apache.tools.ant.util.SymbolicLinkUtils.isSymbolicLink(
      SymbolicLinkUtils.java:107)
3   at org.apache.tools.ant.util.SymbolicLinkUtils.isSymbolicLink(
      SymbolicLinkUtils.java:73)
4   at org.apache.tools.ant.util.SymbolicLinkUtils.deleteSymbolicLink(
      SymbolicLinkUtils.java:223)
5   at org.apache.tools.ant.taskdefs.optional.unix.Symlink.delete(Symlink.java
      :187)
```

Listing 2.3: Crash Stack Trace for ANT-49137.

As Listing 2.4 depicts, the test case by EvoCrash creates an instance of `Symlink`, `symlink0`, adapts the state in `symlink0`, and ultimately makes a call to `delete`, which will result in generating the target crash stack trace with fitness equal to 0.0. On the other hand, as Listing 2.5 shows, the test case by STAR, makes an instance of `SymbolicLinkUtils`, which comes before the defective frame in the crash stack, and makes a call to the root method, `isSymbolicLink`. Consequently, only part of the target crash stack is generated by this test, which is shown in Listing 2.6. Since the defective frame is not revealed in the resulting crash trace, even though the root frame is covered, the test by STAR does not evaluate to useful according to the criteria

[12] `https://bz.apache.org/bugzilla/show_bug.cgi?id=49137`

set by STAR [81].

```
public void test0() throws Throwable {
 Symlink symlink0 = new Symlink();
 symlink0.setLink("");
 symlink0.delete();
}
```

Listing 2.4: Generated test by EvoCrash for ANT-49137.

```
public void test0() throws Throwable {
   java.io.File v1 = (java.io.File) null;
   org.apache.tools.ant.util.SymbolicLinkUtils v2 =
       org.apache.tools.ant.util.SymbolicLinkUtils.getSymbolicLinkUtils();
   v2.isSymbolicLink((java.io.File) v1, (java.lang.String) null);
}
```

Listing 2.5: Generated test by STAR for ANT-49137.

```
1 java.lang.NullPointerException
2   at org.apache.tools.ant.util.SymbolicLinkUtils.isSymbolicLink(
      SymbolicLinkUtils.java:107)
```

Listing 2.6: Generated Crash Stack Trace by STAR for ANT-49137.

Other than ACC-104, ANT-43292 is the other case that is only reproducible by STAR. The main reason for this lies in an inheritance-related problem and how the current fitness function compares stack frames. In this case, the target method, `mapFileName`, is defined in `FilterMapper`, which extends `FileNameMapper`. However, the search can find better fitness values, using other subclasses of `FileNameMapper`, such as `FlatFileNameMapper`, because the implementation of `mapFileName` in these subclasses has lower complexity.

For LOG, EvoCrash covers all the cases that were covered by STAR. Six of the LOG cases (33%) are only covered by EvoCrash. As an example, for LOG-509 there is a need to interact with the file system in order to open a file, and in order to do so, EvoCrash benefits from the mocking mechanisms implemented in EvoSuite.

LOG-47912 (shown in Listing 2.7) is another example for which only EvoCrash successfully generated a useful test case. The buggy frame in this case is at level four, and the generated test by EvoCrash is at level five, which is shown in Listing 2.8. As the listing shows, in order to generate a test at this level, several complex objects need to be generated and set up first, until finally the call to *jULBridgeHandler0.publish(logRecord0);* is made. This example shows the capability of EvoCrash to

generate complex objects which may be needed to execute a particular execution path that leads to the target line where the target exception is thrown.

```
1 java.lang.NullPointerException:
2     at org.apache.log4j.CategoryKey.(CategoryKey.java:32)
3     at org.apache.log4j.Hierarchy.getLogger(Hierarchy.java:266)
4     at org.apache.log4j.Hierarchy.getLogger(Hierarchy.java:247)
5     at org.apache.logging.julbridge.JULLog4jEventConverter.convert(
      JULLog4jEventConverter.java:121)
6     at org.apache.logging.julbridge.JULBridgeHandler.publish(
      JULBridgeHandler.java:49)
```

Listing 2.7: Stack Trace for LOG-47912.

```java
public void test0() throws Throwable {
   Logger logger0 = Logger.getLogger("I}h}$.Xa|yA,YSXf");
   Hierarchy hierarchy0 = (Hierarchy)logger0.getLoggerRepository();
   JULLog4jEventConverter jULLog4jEventConverter0 = new
       JULLog4jEventConverter((LoggerRepository) hierarchy0,
       (JULLevelConverter) null);
   JULBridgeHandler jULBridgeHandler0 = new
       JULBridgeHandler((LoggerRepository) hierarchy0,
       jULLog4jEventConverter0);
   Level level0 = Level.SEVERE;
   LogRecord logRecord0 = new LogRecord(level0, "");
   jULBridgeHandler0.publish(logRecord0);
}
```

Listing 2.8: Generated test by EvoCrash for LOG-47912.

EvoCrash *vs.* MuCrash. As Table 2.4 shows, evaluation data for MuCrash is only available for ACC.[13] Except for ACC-104, EvoCrash covers all the ACC-cases that are covered by MuCrash. In addition, three cases (25%) are only covered by EvoCrash, though one of them is not marked as useful.

An example of a covered case is ACC-53, depicted in Listing 2.9. It requires that an object is added to an instance of `UnboundedFifoBuffer`, the `tail` index is set to a number larger than the buffer size, and then that the method `remove` is invoked. In addition, the order in which the methods are invoked matters. So, if the `tail` index would be set after `remove` is called, the target crash would not be replicated. As shown in Listing 2.10, EvoCrash synthetized the right method sequence and reproduced ACC-53.

[13] Since MuCrash is not publicly available we could not reproduce the data or add additional cases by ourselves.

```
java.lang.ArrayIndexOutOfBoundsException:
  at org.apache.commons.collections.buffer.UnboundedFifoBuffer$1.remove(
    UnboundedFifoBuffer.java:312)
```

Listing 2.9: Crash Stack Trace for ACC-53

```
Object object0 = new Object();
UnboundedFifoBuffer unboundedFifoBuffer0 = new UnboundedFifoBuffer();
unboundedFifoBuffer0.add(object0);
unboundedFifoBuffer0.tail = 82;
unboundedFifoBuffer0.remove((Object) null);
```

Listing 2.10: EvoCrash test for ACC-53

EvoCrash *vs.* JCHARMING. As Table 2.4 shows, we have 12 cases to derive comparisons between EvoCrash and JCHARMING. While 75% of the cases are covered both by EvoCrash and JCHARMING, there is substantial difference in the efficiency of the two approaches. On average, EvoCrash takes less than 2 minutes to cover the target crashes, whereas (based on the published results) JCHARMING may take from 10 to 38 minutes to generate tests for the same cases.

Among the LOG cases, two out of seven (29%) are only supported by EvoCrash. As an example, Listing 2.11 shows the crash stack trace for LOG-45335, which is one of the two cases covered only by EvoCrash. To generate a useful test for LOG-45335, as depicted in Listing 2.12, EvoCrash sets the `ht` state in `NDC` to `null`, and then makes a call to the static method `remove`, which is the buggy frame method.

Among the other cases, two of them are only supported by EvoCrash, `ANT-41422`, and `ActiveMQ-5035`. The former is a `NullPointerException`, and the latter is a `ClassCastException`.

```
java.lang.NullPointerException:
 at org.apache.log4jb.NDC.remove(NDC.java:377)
```

Listing 2.11: Crash Stack Trace for LOG-45335.

```
public void test0() throws Throwable {
    NDC.ht = null;
    NDC.remove();
 }
```

Listing 2.12: The EvoCrash Test for LOG-45335.

2.4.7 Threats to Validity

In this section, we outline various possible threats to the validity of the empirical evaluation we conducted.

External Validity. The main threats arise from the focus on Java and open source projects. The use of Java is needed for our experiments due to the dependency on EvoSuite, yet we expect our approach to behave similarly on other languages such as Ruby or C#.

To maximize reproducibility and to enable comparison with the state-of-the-art we rely on open source Java systems. We see no reason why closed-source stack traces would be substantially different. As part of our future work, we will engage with one of our industrial partners, mining their log files for frequent stack traces. This will help them create test cases that they can add to their test suite to reproduce and fix errors their software suffers from.

To facilitate comparison with earlier approaches, we selected bugs and system versions that have been used in earlier studies, and hence are several years old. We anticipate that our approach works equally-well on more recent bugs or versions as well, but have not conducted a systematic experiments yet.

A finding of our experiments is that a key limiting factor for any stack-trace based approach is the unavailability of external data that may be needed for the reproduction. Further research is needed to (1) mitigate this limitation; and (2) identify a different data set of crashes focusing on such missing data, in order to further narrow down this problem.

Internal Validity. A key threat to the internal validity is in the evaluation of the crash coverage and usefulness of the generated test cases. In case EvoCrash generated a test with fitness = 0.0, we rerun the generated test against the SUT to double checked that the generated crash stack trace correctly replicated the target crash stack. Despite having taken the above procedures, it is still possible that we made errors in the inspections and evaluations. To mitigate the chances of introducing errors, we peer-reviewed tests and crashes. In addition, we make the EvoCrash tool, and the generated test cases publicly available for further evaluations.

2.5 Study II: Usefulness for Debugging

To assess the degree to which generated crash-reproducing tests are useful during debugging, we conduct a controlled experiment. The experiment aims to address the following:

- **RQ4:** *Do participants who use EvoCrash tests more often locate defects compared to participants who do not use EvoCrash tests?* With this research question, we aim to understand whether using the generated tests by EvoCrash helps locate defects.
- **RQ5:** *Do participants who use EvoCrash tests more often provide fixes compared to participants who do not use EvoCrash tests?* With this research question, we aim to investigate whether using the generated test by EvoCrash helps fixing defects.
- **RQ6:** *Do participants who use EvoCrash tests spend less time than participants who do not use EvoCrash tests?* With this research question, we aim to analyze the impact of using the generated tests by EvoCrash in the amount of time the participants took to deliver fixes.

2.5.1 Task Selection

To select the crash cases to be used in the debugging tasks, we considered the following selection criteria: (i) From the 54 crashes we used in the empirical evaluation (Section 4.4), we selected those crashes which signal the two common types of exceptions in Java programs [84], namely: `NullPointerException`, and `IllegalArgumentException`; (ii) We filtered out stack traces which have less than four stack frames, since locating and fixing the related bug would be very simple; (iii) To avoid cases that would be overly complicated to fix, we selected cases for which the original fixes (delivered by the original developers) are provided for the classes that were included in the stack traces. (iv) We ensured that the JavaDoc documentation is available for all classes appearing in the stack traces and could serve as specification for the participants. Finally, (v) considering the usefulness criterion (described in Section 2.4.1), we opted for including both a *useful* and *not useful* crash-reproducing unit test case.

As the result, we selected ACC-48 (with a useful test), and LOG-47957 (with a not useful test) to be the target cases. Listing 2.13 and 2.14 show the stack traces for the two cases.

```
java.lang.IllegalArgumentException: Initial capacity must be greater than 0
   at org.apache.commons.collections.map.AbstractHashedMap.
     (AbstractHashedMap.java:142)
   at org.apache.commons.collections.map.
     AbstractHashedMap.(AbstractHashedMap.java:127)
   at org.apache.commons.collections.map.AbstractLinkedMap.
   (AbstractLinkedMap.java:95)
   at org.apache.commons.collections.map.LinkedMap.
     (LinkedMap.java:78)
   at org.apache.commons.collections.map.TransformedMap.
     transformMap(TransformedMap.java:153)
   at org.apache.commons.collections.map.TransformedMap.putAll
     (TransformedMap.java:190)
```

Listing 2.13: Crash Stack Trace for ACC-48; Fixed at frame 5 (line 153) and tested at frame 6 (line 190).

```
java.lang.NullPointerException:
   at org.apache.log4jb.net.SyslogAppender.append(SyslogAppender.java:251)
   at org.apache.log4jb.AppenderSkeleton.doAppend(AppenderSkeleton.java:230)
   at org.apache.log4jb.helpers.AppenderAttachableImpl.appendLoopOnAppenders
     (AppenderAttachableImpl.java:66)
   at org.apache.log4jb.Category.callAppenders(Category.java:203)
   at org.apache.log4jb.Category.forcedLog(Category.java:388)
   at org.apache.log4jb.Category.info(Category.java:663)
```

Listing 2.14: Crash Stack Trace for LOG-47957; Fixed and tested at frame 1 (line 251).

The original fixes for `ACC-48` and `LOG-47957` were provided for the frame levels five and one, respectively. In addition, the tests from EvoCrash for these cases were targeted for the frame levels six and one, respectively.

2.5.2 Experiment Participants

We invited 35 master students in computer science from the Delft University of Technology to participate in the study. Table 2.5 presents the level of formal education the participants have in Java programming. Table 2.6 presents the degree to which the participants have industrial experience in software engineering. Moreover, Table 2.7 summarizes the degree to which the participants were familiar with the JUnit testing framework.

Table 2.5: Participants' Education in Java Programming

Self-educated	Formal Education		
	Basic	Intermediate	Advanced
5.71%	28.57%	45.71%	20%

Table 2.6: Participants' Industrial Experience

No exp.	≤ 2 years	3-5 years	5-10 years
42.85%	34.28%	20%	2.85%

2.5.3 Experiment Procedure

Before conducting the experiment, the participants received an introduction to the tasks to perform. The students had two weeks within which, at some point they were to start performing the experiment and deliver the results. Notice that to avoid any bias, we made sure participants were neither aware of the research questions of our study nor which crashes (name and id) were used as subjects of the experiment.

The participants were asked to debug and fix the classes involved in the two bugs ACC-48, and LOG-47957 starting from the corresponding crash stack traces. Each participant had to perform one bug fixing task using the crash-reproducing test from EvoCrash (e.g., ACC-48), while for the other one (e.g., LOG-47957) we did not provide the test from EvoCrash. To address potential bias due to learning effects, we assigned the tasks to have a balanced number of participants that performed the first task with and without the EvoCrash test. Therefore, we randomly grouped students in four different groups, whose configurations are shown in Table 2.8.

Once participants started performing the experiment at some point within the two weeks, they were asked to complete three stages in the context of the experiment: (i) filling a pre-test questionnaire that we used to collect data about participants' background, (ii) performing the first debugging task and filling the corresponding post-test questionnaire, and (iii) performing the second debugging task and filling a second post-test questionnaire. While the time to complete the first stage was unbounded, for the remaning two stages we restricted the amount of time participants could spend on each task following the guidelines by [212]. In particular, participants had 45 minutes for each task, which includes: (i) reading the instructions, (ii) cloning a Maven project from GitHub, and (iii) performing the corresponding debugging task. Each debugging task consists of (i) locating the defect that trigger the target crash, (ii) providing the code fix, (iii) running the existing test suite and adding new tests if needed. The participants could finish the tasks in less than 45 minutes if they were

Table 2.7: Participants' Familiarity with the JUnit Framework

Unfamiliar	Basic	Average	Advanced
37.14%	25.71%	28.57%	8.57%

Table 2.8: Configuration of the Participant Groups

Group	Task 1		Task 2	
	Bug	EvoCrash	Bug	EvoCrash
I	ACC-48	Yes	LOG-47957	No
II	ACC-48	No	LOG-47957	Yes
III	LOG-47957	Yes	ACC-48	No
IV	LOG-47957	No	ACC-48	Yes

sure that (i) the identified bug location is correct, and (ii) the provided fixes prevent the crashes to incur again and do not break the existing test suite. Controlling the time allowed to prevent that too little or too long time would be spent by participants on each task.

To prepare the projects on GitHub, we selected the versions of `Apache Commons Collections`, and `Apache Log4j` that were specified in the bug reports for ACC-48 and LOG-47957. Both projects were already Maven projects, so we imported them into Eclipse, and made sure the tests were run with no particular difficulties. For those tasks where the test from EvoCrash was provided, we included the tests in the projects, and added their path (packages) in the instructions provided to the participants.

As the first task reached the time out, or the participants completed the task within 45 minutes, they would proceed to the follow-up post-test questionnaire. To make sure the participants do not take time at this point to keep working on the task, we allowed 10 minutes to be spent on answering the questions. The second task followed the same procedure as the first one, after which the assignment would be completed. At the end, the participants had to send the artifacts they produced (including any test cases, or fixes) via e-mail to the first author. Furthermore, we used the online platform: *https://www.qualtrics.com* to collect the results of the questionnaires.

Before conducting the experiment, the last two authors performed the tasks to assess their feasibility and correctness in advance. We also conducted three pilot studies with external researchers within the software engineering research group at Delft University of Technology. The feedback we received from the pilot studies were used to improve both the questionnaires and the instructions for the tasks. Data points from dry-runs and pilot studies are not included in our analysis of the results.

2.5.4 Data Analysis

The original location of the defects and the patches provided by the developers for both cases represent our *golden answers* (oracle) as for the defect locations and fixes.

To answer **RQ**3, we compared the bug locations which were pointed to by the participants with the locations in our golden set. For example, for ACC-48, the defect could be fixed at two different frame levels in the stack trace, namely: (a) in the `transformMap` method at the 5th frame level in the stack trace reported in Listing2.13, and (b) in the `putAll` method at the 6th level. However, it is important to target the `transformMap` routine as the location for the underlying defect, and not the `putAll` routine. This is because `putAll` is an API call whereas `transformMap` is a private routine to which other routines make calls as well. Therefore, `transformMap` is the root location where the defect must be fixed otherwise the crash could recur. In cases where the participants targeted `putAll` as the buggy location, we marked their answers as incorrect.

For what concerns **RQ**4, we ran the fixes given by the participants to assess whether they prevented the crashes from recurring. If so, then we manually analyzed the content of the fixes. For example, in case of the fixes given for LOG-47957, we accepted every fix which pointed to checking for `null` references at the right location in the source code.

For what regards **RQ**5, we utilized the data that was provided by the online platform for collecting the data related to the time participants took to deliver the fixes. The data measured the point in time when the participant started a task (by reading the instructions), and the point in time when the participant completed the task before proceeding to answering the subsequent questions.

2.5.5 Statistical Analysis

To assess the effect of using the EvoCrash tests on the ability of participants to locate and fix the defects, we used the *odds ratio* measure [50] since the data is binary distributed, i.e., the defect is correctly located (or fixed) or not. For this test, we use a 95% confidence interval and we computed it for each debugging task (ACC-48, and LOG-47957) separately. In addition, to determine the significance of the findings, we used the Fisher's exact test, which is can be used for small sample sizes [50]. We considered $\alpha = 0.05$ for the Type I error. Significant p-values (i.e., lower than 0.05) indicate that participants with EvoCrash tests were able to correctly locate and

fix defects more frequently compared the participants who performed the same task (e.g., ACC-48) without the EvoCrash tests.

To measure the effect of using EvoCrash tests on the amount of time the groups took to complete each task, we used the Vargha-Delaney $\hat{A}$ statistic [207]. We selected this effect size measure since it is well-suited for numerical data distributions [207], such as the time in seconds. Values of $\hat{A}<0.50$ indicate that participants with the EvoCrash tests spend less time than the participants without the EvoCrash tests to complete the same task; values of $\hat{A}>0.50$ indicates the opposite scenario, i.e., participants with the EvoCrash tests spent more time to complete the assigned tasks; $\hat{A}=0.50$ when there is no difference between the participants who performed the tasks with and without EvoCrash. The effect size can be classified as one of the four different levels [207]: *negligible* ($\hat{A}\geq0.44$), *small* ($0.36\leq\hat{A}<0.44$), *medium*($0.29\leq\hat{A}<0.36$), or *large* ($\hat{A}\leq0.29$). For a given task, we also test whether the difference (if any) between the groups with and without EvoCrash were statistically significant by using the non-parametric Wilcoxon Rank Sum test with $\alpha = 0.05$ for the Type I error. Significant p-values imply that there is significant difference in the amount of time the participants take when performing the debugging tasks with and without EvoCrash.

2.5.6 Analysis of the Results

In this section, we present the results of the controlled experiment with student participants. Table 2.9 summarizes the results regarding assessing the impact of using the tests from EvoCrash on the ability of the participants in locating the defects and providing fixes for them. As Table 2.9 indicates, one of 35 students, corresponding to one of the groups II or III in Table 2.8, did not deliver the debugging tasks. Thus, the number of participants in these groups is 34. On the other hand, all participants corresponding to groups I and IV in Table 2.8 delivered the debugging tasks. Therefore, the total number of participants in Table 2.9 is 35. In what follows, we discuss the results and thereby answer RQ_4, RQ_5, and RQ_6, respectively.

Table 2.9: Results of RQ_4 and RQ_5 grouped by tasks ("With" = with EvoCrash tests, "Without" = without EvoCrash tests).

Metrics	ACC-48		LOG-47957	
	With	Without	With	Without
No. of correct bug locations	13	13	12	9
No. of incorrect bug locations	5	3	4	8
No. of undelivered tasks	0	1	1	0
No. of correct bug fixes	10	3	8	6
No. of incorrect bug fixes	8	13	8	11
No. of undelivered tasks	0	1	1	0

2.5.6.1 RQ_4: Impact of EvoCrash Tests on Locating Defects

As Table 2.9 shows, in the case of `ACC-48`, the number of participants who located the defect correctly, using the test from EvoCrash, is the same as the number of participants who did not use the test from EvoCrash. The number of participants who failed to locate the defect, with and without using the EvoCrash test, are five and three, respectively. In the case of `LOG-47957`, 12 participants, using the test from EvoCrash, and nine participants without using the test correctly located the defect. The number of participants who failed to locate the defect, with and without the EvoCrash test, are four and eight, respectively.

To assess the impact of using EvoCrash on locating the underlying defects for each of the debugging task, we used the odds ratio and Fisher's exact test as explained in Section 2.5.5. For `ACC-48`, the odds ratio is 0.63, thus, indicating that the test case from EvoCrash did not help the participants in locating the underlying defect. Moreover, the Fisher test further confirms that there is no statistically significant difference between the two groups (p-value = 0.86). For `LOG-47957`, the odds ratio is 2.66, suggesting that the test from EvoCrash helped the participants in locating the underlying defect more often than the participants who did not use the test. However, these results are not statistically significant in this case either (p-value=0.14).

> **RQ_4:** EvoCrash helps participants in locating the defect for `LOG-47957`, while in the case of `ACC-48` we did not observe such an impact. In either case, the differences are not statistically significant.

2.5.6.2 RQ_5: Impact of EvoCrash Tests on Fixing Defects

As Table 2.9 shows, in the case of `ACC-48`, the number of participants who provided acceptable fixes are 10 when using the test from EvoCrash and three without the test. In addition, eight and 13 participants, with and without the test from EvoCrash respectively, failed to provide an acceptable fix for `ACC-48`. In the case of `LOG-47957`, eight participants, using the test from EvoCrash, and six participants without using the generated test provided acceptable fixes. The number of participants who failed to provide acceptable fixes, with and without using the test is eight and 11, respectively.

To assess the impact of using EvoCrash on the ability of participants in providing fixes, we computed the odds ratio for each debugging task, separately. In addition, we used the Fisher's exact test for significance.

For ACC-48, the odds ratio is 5.41. This indicates that the test case generated by EvoCrash increased the participants' ability to provide fixes when performing such a debugging task. According to the Fisher test, the differences are statistically significant (p-value=0.03). We further note, based on the usefulness criterion described in Section 2.4.1 we labeled the test generate by EvoCrash as useful for debugging.

For LOG-47957, the odds ratio is 1.83. Based on these measures, we observed that using the test from EvoCrash increased the participants' ability to provide correct fixes. However, such an improvement is not statistically significant as suggested by the Fisher test (p-value=0.30). These results are in line with the results of Study I, where we labeled the test generated by EvoCrash as *not useful* according to the usefulness criterion described in Section 2.4.1).

RQ$_5$: Using a test from EvoCrash, that is *useful* according the usefulness criterion in Section 2.4.1, increases developers' ability in fixing defects when debugging. In addition, our results suggest that using a test from EvoCrash, that is *not useful* according to the usefulness criterion in Section 2.4.1, also increases developers' ability in fixing defects when debugging. However, in the latter case, the difference is not statistically significant.

2.5.6.3 RQ$_6$: Impact of EvoCrash Tests on Debugging Time

The box-plots in Figure 2.3 show the distribution of time participants took to perform each task. In the case of ACC-48, the median for the group which did not use the EvoCrash test is 1565 seconds, while the median for the other group, using the EvoCrash test, is 1064 seconds (-32%). In the case of LOG-47957, the median for the group which did not use the EvoCrash test is 2700 seconds, while the median for the other group, using the EvoCrash test, is 2037 seconds (-25%). Thus, in both cases, the medians for the group which used the tests from EvoCrash are lower than the median for the group which did not use the EvoCrash tests.

To verify whether such differences are statistically significant or not, we used the non-parametric Wilcoxon test for each debugging task (ACC-48, and LOG-47957) as described in Section 2.5.5. As effect size measure, we used the Vargha-Delaney $\hat{A}$ statistics.

For ACC-48 and LOG-47957, the obtained $\hat{A}$ scores are 0.28 (medium) and 0.30 (medium), respectively. The differences between the groups with and without the

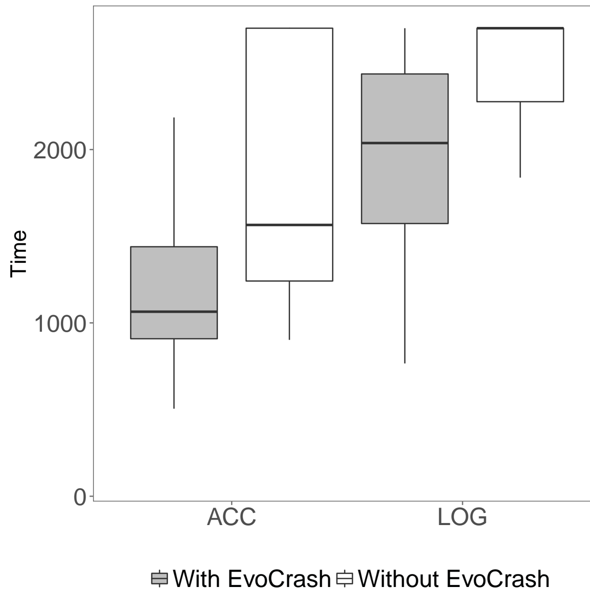


Figure 2.3: Amount of time participants took to perform each task, with and without the tests from EvoCrash.

EvoCrash test are also statistically significant according to the Wilcoxon test, which returns p-values of 0.03 and 0.04 for `ACC-48` and `LOG-47957`, respectively. Based on the results above, we conclude:

> **RQ$_6$:** Developers using the tests from EvoCrash take significanlty less time when debugging, compared to those not using the EvoCrash tests.

2.5.7 Threats to Validity

In this section, we outline various possible threats to the validity of the controlled experiment we conducted.

Internal Validity. To reduce factors that could affect the causal relations under scrutiny, we randomly assigned the tasks to the participants. Regarding the ability of the participants in locating the defects and fixing them, it could be that not being familiar with the source code negatively affects the degree to which the participants were able

to locate and fix the defects. To mitigate this impact: i) We made sure Java-Doc documentation is available for the target projects to be debugged, and ii) We checked with the pilot studies whether the given time for each task was reasonable, and whether the available documentation was sufficient to perform them.

Moreover, we conducted the experiment remotely from the participants, which implies that they would do the experiment at their own discretion. Using the online platform, we made sure the participants are mandated to perform the tasks in the specified order, and within the specified time limit. In addition, the participants could only answer each follow up questionnaire after they had completed each task.

External Validity. One factor that could affect the generalizability of the study could be the student participants of the experiment. Different studies [124, 166] show that if students are familiar with performing the tasks of the experiment, then they would perform similar to participants from industry. Over 50% of the participants declared to have at least 2 years of industrial experience, and basic familiarity with the JUnit framework. In addition, by giving an introductory lecture we further tried to familiarize the students and thereby, mitigate possible threats to the generalizability of the experiment results.

Furthermore, we analyzed only two types of exceptions in the experiment. As described in Section 2.5.1, to select these types we considered a number of criteria, including how often they occur, the stack trace sizes, and whether they are overly complex or overly simple cases to debug. We deliberately opted for only two exceptions in order to i) maintain statistical power in the analysis, and ii) avoid introducing fatigue and learning effects to the participants.

Construct Validity. Threats to this type of validity concern the degree to which the conducted experiment measures what is intended to be measured. We used the online platform to measure the amount of time each participant took to complete the debugging tasks. Since the experiment was done remotely, we did not fully observe how the participants spent the debugging time they took. While by limiting the debugging time and providing the questions after each task was completed we tried to control the experiment flow, it is possible that the participants did not spend the entire time on the debugging tasks.

Conclusion Validity. We conducted the experiment with 35 master students. In the experiment, each task was performed by at least 16 students. While 16 is not a large number as for the size of each group, it still yields sufficient statistical power to assess the impact of using EvoCrash tests on the number of fixed bugs (when the test is useful for debugging), as well as the amount of time it takes to finish the debugging tasks. Regarding assessing the impact of EvoCrash tests on the ability of developers

in locating defects, our experiment shows preliminary results, and therefore indicates the need for further future investigation.

We support our findings by using appropriate statistical tests (namely: The Fisher's exact test, odds ratio measure, Wilcoxon Rank Sum test, and the Vargha-Delaney $\hat{A}$ statistic) to assess the impact of using EvoCrash tests in debugging.

2.6 Discussion and Lessons Learnt

Interactive Search. It should be noted that since GGA strives for finding the fittest test case, thus discarding the ones with fitness > 0.0, the crash coverage and usefulness evaluation was performed on a set of EvoCrash tests with fitness equal to 0.0. However, considering the crash exploitability and usefulness criteria adopted from STAR [81], it could be possible that EvoCrash discarded tests with fitness between 0.0 and 1.0, which would actually conform to the aforementioned criteria. Considering the fitness function range, fitness values could be from 0.0 to 6.0, where 6.0 means a test case that does not reach the target line, therefore does not invoke the target method, and in turn, does not trigger the target exception. In contrast, fitness 0.0 means that the test covers the target line and method, and triggers the target exception. According to the definition of the fitness function (presented in Section 4.2.2), when the fitness value is between 0.0 and 1.0, the target line and exception are covered, however, the stack trace similarity is not ideal yet. In this case, even though the target stack similarity is not achieved, crash coverage and test usefulness criteria could be covered. Future work can provide interactive mechanisms through which the precision of the fitness function could be adjusted, so tests with fitness between 0.0 and 1.0 could also be accepted.

In addition, dependency on external files was a major factor that prevented EvoCrash from covering more cases. Therefore, if external files were to be provided by the bug reporters, then enabling developers to specify the external files could be another possible direction for the future work.

Extending Comparisons. Towards extending the empirical evaluation, we aimed at adopting the crash cases reported in [172] in order to make a larger comparison with JCHARMING. However, due to various reasons, ultimately we managed to adopt four cases to this end. While the new cases provide a bigger picture, we are still interested to expand the comparisons among the recent tools for automated crash reproduction. This aim would be facilitated if the tools become publicly available.

In addition, we acknowledge the need for extending the empirical evaluation of EvoCrash to crashes from recent industrial projects. In such projects scale, complexity, and type of the generated crashes may vary, which may indicate new research dimensions to consider for improving our search-based crash reproduction approach.

Controlled Experiment. To analyze the impact of EvoCrash tests in debugging, we selected two common exceptions in Java programs, `NullPointerException`, and `IllegalArgumentException`. This is while various types of exceptions may impose different levels of complexity in debugging, and thus, the impact of crash reproducing tests may vary in each case. Therefore, future studies could adopt more common exceptions in Java programs, and assess the impact of EvoCrash tests per exception type.

In addition, our experiment results showed that using *useful* EvoCrash tests helps developers fix bugs and take less time in debugging. While using such tests helped the participants locate the given defect, the observed impact was not statistically significant. To be able to locate the root cause of a given failure, having upfront understanding and knowledge about the defective source code may be another important factor that can impact the ability of a developer in localizing a given defect. Therefore, future studies may assess the impact of having up-front knowledge of source code and its correlation with using crash reproducing tests in debugging.

2.7 Conclusions

Several approaches to automated crash replication have been proposed to aid developers when debugging. However, these approaches report several challenges such as path explosion and handling environmental dependencies in practice. We propose a new approach, EvoCrash, to automated crash reproduction, via a Guided Genetic Algorithm (GGA). Our empirical evaluation on 54 real-world crashes shows that GGA addresses the path explosion problem. Furthermore, thanks to the mocking mechanisms in EvoSuite, some crashes involving environmental interactions were reproduced. However, handling environmental dependencies (such as content of a required file) remain to be a challenge for EvoCrash. We acknowledge the need for further empirical evaluations on more recent and industrial cases. The result of such evaluations may help identify the areas where we can improve our search-based crash reproduction technique.

In addition, we compare effectiveness and efficiency of EvoCrash with EvoSuite as a whole test suite generation approach to coverage-based test generation. Our results

confirm that the provided guidance in GGA is necessary for effectively and efficiently reproducing the crashes.

Moreover, we report from a controlled experiment with 35 master students in computer science, in which we assessed the impact of using EvoCrash tests in practice. Based on the results of the controlled experiment, we observed that: i) Our data regarding the impact of EvoCrash tests on the ability of developers in locating defects is preliminary. Therefore, our results show need for further future investigation in this regard. ii) Using a useful test from EvoCrash when debugging, developers can provide fixes more often, compared to when debugging without using such tests. Finally, iii) using EvoCrash tests reduces the amount of time developers take when debugging.

Algorithm 2.1 Guided Genetic Algorithm

Input: Class under test C

Target call from the crash stack trace TC

Population size N

Search time-out max_time

Result: Test case t

```
1  begin
2      // initialization  M_crash ⟵ identify public methods based on TC
3      k ⟵ 0
4      P_k ⟵ MAKE-INITIAL-POPULATION(C, M_crash, N)
5      EVALUATE(P_k)
6      // main loop
7      while (best fitness value > 0) AND (time spent < max_time) do
8          k ⟵ k + 1  // generate offsprings
9          O ⟵ Ø
10         while | O |< N do
11             p_1, p_2 ⟵ select two parents for reproduction
12             if crossover probability then
13                 o_1, o_2 ⟵ GUIDED-CROSSOVER(p_1, p_2)
14             else
15                 o_1 ⟵ p1
16                 o_2 ⟵ p2
17             O ⟵ O ⋃ GUIDED-MUTATION(o_1)
18             O ⟵ O ⋃ GUIDED-MUTATION(o_2)
19         // fitness evaluation
20         EVALUATE(O)
21         P_k ⟵ P_{k−1} ⋃ O
22         P_k ⟵ select the N fittest individuals in P_k
23     t_best ⟵ fittest individual in P_k
24     t_best ⟵ POST-PROCESSING(t_best)
```

Algorithm 2.2 MAKE-INITIAL-POPULATION

```
Input: Class under test C
       Set of failing methods M_crash
       Population size N
Result: An initial population P_0
25 begin
26   P_0 ⟵ ∅
27   while | P_0 |< N do
28     t ⟵ empty test case  size ⟵ random integer ∈ [1;MAX_SIZE]
29     // probability of inserting a method involved in the failure
30     insert_probability ⟵ 1/size
31     while (number of statements in t) < size do
32       if random_number ⩽ insert_probability then
33         method_call ⟵ pick one element from M_crash
34         // reset the probability of inserting a failing method
35         insert_probability ⟵ 1/size
36       else
37         method_call ⟵ pick one public method in C
38         length ⟵ number of statements in t  // increase the probability of inserting a failing method
39         insert_probability ⟵ 1/(size − length + 1)
40       INSERT-METHOD-CALL(method_call, t)
41     P_0 ⟵ P_0 ⋃ t
```

Algorithm 2.3 GUIDED-CROSSOVER

Input: Parent tests p_1 and p_2

Set of failing methods M_{crash}

Result: Two offsprings o_1, o_2

```
42 begin
43     size_1 ⟵| p_1 |
44     size_2 ⟵| p_2 |
45     // select a cut point
46     μ ⟵ random number ∈ [0; 1]
47     // first offspring
48     o_1 ⟵ first μ × size_1 statements from p_1
49     o_1 ⟵ append (1 − μ) × size_2 statements from p_2
50     CORRECT(o_1)
51     if o_1 does not contain methods from M_crash then
52         o_1 ⟵ clone of p_1
53     // second offspring
54     o_2 ⟵ first μ × size_2 statements from p_2
55     o_2 ⟵ append (1 − μ) × size_1 statements from p_1
56     CORRECT(o_2)
57     if o_2 does not contain methods from M_crash then
58         o_2 ⟵ clone of p_2
```

Algorithm 2.4 GUIDED-MUTATION

Input: Test $t = \langle s_1, \ldots, s_n \rangle$ to mutate

Set of failing methods M_{crash}

Result: Mutated test t

```
59 begin
60     n ⟵| t |
61     apply_mutation ⟵ true
62     while apply_mutation == true do
63         for i =1 to n do
64             φ ⟵ random number ∈ [0; 1]
65             if φ ⩽ 1/n then
66                 if delete probability then
67                     delete statement s_i
68                 if change probability then
69                     change statement s_i
70                 if insert probability then
71                     insert a new method call at line i
72         if t contains method from M_crash then
73             apply_mutation ⟵ false
```

3

Large-scale Evaluation of EvoCrash

Crash reproduction approaches help developers during debugging by generating a test case that reproduces a given crash. Several solutions have been proposed to automate this task. However, the proposed solutions have been evaluated on a limited number of projects, making comparison difficult. In this paper, we enhance this line of research by proposing JCrashPack, an extensible benchmark for Java crash reproduction, together with ExRunner, a tool to simply and systematically run evaluations. JCrashPack contains 200 stack traces from various Java projects, including industrial open source ones, on which we run an extensive evaluation of EvoCrash, the state-of-the-art tool for search-based crash reproduction. EvoCrash successfully reproduced 43% of the crashes. Furthermore, we observed that reproducing NullPointerException, IllegalArgumentException, and IllegalStateException is relatively easier than reproducing ClassCastException, ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException. Our results include a detailed manual analysis of EvoCrash outputs, from which we derive 14 current challenges for crash reproduction, among which the generation of input data and the handling of abstract and anonymous classes are the most frequents. Finally, based on those challenges, we discuss future research directions for search-based crash reproduction for Java.

3.1 Introduction

Software crashes commonly occur in operating environments and are reported to developers for inspection. When debugging, reproducing a reported crash is among the

tasks a developer needs to do in order to identify the conditions under which the reported crash is triggered [221]. To help developers in this process, various automated techniques have been suggested. These techniques typically either use program *runtime data* [**?**, 58, 64, 76, 114, 169, 194, 205] or crash *stack traces* [65, 81, 173, 202, 215] to generate a test case that triggers the reported crash.

When available, runtime data offer more information to accurately reproduce a crash. However, it also raises various concerns (for instance, privacy violation) and may induce a significant overhead during data collection [81,173,194]. Instead, we focus on crash reproduction based on a crash stack trace generated by a failing system. Practically, those stack traces are collected from the logs produced by the operating environment or reported by users in an issue tracking system. Various auromated crash stack trace-based reproduction approaches have been implemented and evaluated on different benchmarks [81,173,202,215]. However, those benchmarks contains a limited number of crashes and associated stack traces.

In a recent study, we presented a search-based approach called EvoCrash, which applies a guided genetic algorithm to search for a crash reproducing test case [202], and demonstrated its relevance for debugging [204]. We conducted an empirical evaluation on 54 crashes from commonly used utility libraries to compare EvoCrash with state-of-the-art techniques for crash reproduction [202]. This was enough to show that the search-based crash reproduction outperformed other approaches based on backward symbolic execution [81], test case mutation [215], and model-checking [173], evaluated on smaller benchmarks.

However, all those crashes benchmarks were not selected to reflect challenges that are likely to occur in real life stack traces, raising threats to external validity. Thus the questions whether the selected applications and crashes were sufficiently representative, if EvoCrash will work in other contexts, and what limitations are still there to address, remained unanswered.

The goal of this paper is to facilitate sound empirical evaluation on automated crash reproduction approaches. To that end, we devise a new benchmark of real-world crashes, called JCrashPack. It contains 200 crashes from seven actively maintained open-source and industrial projects. These projects vary in their domain application and include an enterprise wiki application, a distributed RESTful search engine, several popular APIs, and a mocking framework for unit testing Java programs. JCrashPack is extensible, and can be used for large-scale evaluation and comparison of automated crash reproduction techniques for Java programs.

To illustrate the use of JCrashPack, we adopt it to extend the reported evaluation on EvoCrash [202] and identify the areas where the approach can be improved. In this

experience report, we provide an account of the cases that were successfully reproduced by EvoCrash (87 crashes out of 200). We also analyze all failed reproductions and distill 14 categories of research and engineering limitations that negatively affected reproducing crashes in our study. Some of those limitations are in line with challenges commonly reported for search-based structural software testing in the community [105, 161, 214] and others are specific to search-based crash reproduction.

Our categorization of challenges indicates that environmental dependencies, code complexity, and limitations of automated input data generation often hinder successful crash reproduction. In addition, stack frames (i.e., lines in a stack trace), pointing to varying types of program elements, such as interfaces, abstract classes, and anonymous objects, influence the extent to which a stack trace-based approach to crash reproduction is effective.

Finally, we observe that the percentage of successfully reproduced crashes drops from 85% (46 crashes out of 54 reported by Soltani *et al.* [204]) to 43% (87 out of 200) when evaluating crashes that are from industrial projects. In our observations, generating input data for microservices, and unit testing for classes with environmental dependencies, which may frequently exist in enterprise applications, are among the major reasons for the observed drop in the reproduction rate. These results are consistent with the paradigm shift to context-based software engineering research that has been proposed by Briand et al. [72].

The key contributions of our paper are:

- JCrashPack,[1] a carefully composed benchmark of 200 crashes, as well as their correct system version and its libraries, from seven real-world Java projects, together with an account of our manual analysis on the characteristics of the selected crashes and their constituting frames, including size of the stack traces, complexity measures, and identification of buggy and fixed versions.
- ExRunner,[2] a Python library for automatically running experiments with crash reproduction tools in Java.
- Empirical evidence,[3] demonstrating the effectiveness of search-based crash reproduction on real world crashes taken from JCrashPack.

[1] Available at `https://github.com/STAMP-project/JCrashPack`.
[2] Available at `https://github.com/STAMP-project/ExRunner`
[3] A replication package for EvoCrash results, their automated analysis, and the results of our manual analysis is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application`.

- The identification of 14 categories of research and engineering challenges for search-based crash reproduction that need to be addressed in order to facilitate uptake in practice of crash reproduction research.

The remainder of the chapter is structured as follows: Section *3.2* presents background on crash reproduction. Sections *3.3* to *3.5* describe the design protocol for the benchmark, the resulting benchmark JCrashPack, as well as the ExRunner tool to run experiments on JCrashPack. Sections *3.6* to *3.8* cover the experimental setup for the EvoCrash evaluation, the results from our evaluation, and the results challenges that we identified through our evaluation. Sections *3.9* to *3.12* provide a discussion of our results and future research directions, an analysis of the threats to validity, and a summary of our overall conclusions.

3.2 Background and related work

3.2.1 Crash reproduction

Crash reproduction approaches can be divided into three categories, based on the kind of data used for crash reproduction: *record-replay approaches* record data from the running program; *post-failure approaches* collect data from the crash, like a memory dump; and *stack-trace based post-failure* use only the stack trace produced by the crash. We briefly describe each category hereafter.

Record-replay approaches.

These approaches record the program runtime data and use them during crash reproduction. The main limitation is the availability of the required data. Monitoring software execution may violate privacy by collecting sensitive data, the monitoring process can be an expensive task for the large scale software and may induce a significant overhead [81, 173, 194]. Tools like ReCrash [58], ADDA [**?**], Bugnet [169], jRapture [205], MoTiF [114], Chronicler [64], and SymCrash [76] fall in this category.

Post-failure approaches.

Tools from this category use the software data collected directly after the occurrence of a failure. For instance, RECORE [194] applies a search-based approach to reproduce a crash. RECORE requires both a stack trace and a core dump, produced by the system when the crash happened, to guide the search. Although these tools limit the quantity of monitored and recorded data, the availability of such data still repres-

Table 3.1: The crash stack trace for Apache Ant-49755.

java.lang.**NullPointerException**:

Level	Frame
1	**at** org.apache.tools.ant.util.FileUtils.**createTempFile**(FileUtils.java:**888**)
2	**at** org.apache.tools.ant.taskdefs.TempFile.**execute**(TempFile.java:**158**)
3	**at** org.apache.tools.ant.UnknownElement.**execute**(UnknownElement.java :**291**)

ents a challenge. For instance, if the crash is reported trough an issue tracking system or if the core dump contains sensitive data. Other *post-failure approaches* inlcude: DESCRY [217], and other tools by Weeratunge *et al.* [208], Leitner *et al.* [150, 151], or Kifetew *et al.* [136, 137].

Stack-trace based post-failure.

Recent studies in crash reproduction [65, 81, 173, 202, 215] focuses on utilizing data only from a given crash stack trace to enhance the practical application of the tools. For instance, in contrast to the previously introduced approaches, EvoCrash only considers the stack trace (usually provided when a bug is reported in an issue tracker) and a distance, similar to the one described by Rossler *et al.* [194], to guide the search. Table 3.1 illustrates an example of a crash stack trace from Apache Ant[4] [46] which is comprised of a crash type (`java.lang.NullPointerException`) and a stack of frames pointing to all method calls that were involved in the execution when the crash happened. From a crash stack frame, we can retrieve information about: the crashing method, the line number in the method where the crash happened, and the fully qualifying name of the class where the crashing method is declared.

The state of the research in crash reproduction [65, 81, 134, 173, 202, 215, 219] aims at generating test code that, once executed, produces a stack trace that is as similar to the original one as possible. They, however, differ in their means to achieve this task: for instance, ESD [219] and BugRedux [134] use forward symbolic execution; STAR [81] applies optimized backward symbolic execution and a novel technique for method sequence composition; JCHARMING [173] applies model checking; MuCrash [215] is based on exploiting existing test cases that are written by developers, and mutating them until they trigger the target crash; and Concrash [65] focuses on reproducing *concurrency* failures that violate thread-safety of a class by using search pruning strategies.

[4]ANT-49755: `https://bz.apache.org/bugzilla/show_bug.cgi?id=49755`

3.2.2 Search-based crash reproduction with EvoCrash

Search-based algorithms have been increasingly used for software engineering problems since they are shown to suite complex, non-linear problems, with multiple optimization objectives which may be in conflict or competing [120]. Recently, Soltani et al. [202, 204] introduced a search-based approach to crash reproduction, called EvoCrash. EvoCrash applies a *guided genetic algorithm* to search for a unit test that reproduces the target crash. To generate the unit tests, EvoCrash relies on a search-based test generator called EvoSuite [103].

EvoCrash takes as input a stack trace with one of its frames set as the *target frame*. The target frame is composed of a *target class*, the class to which the exception has been propagated, a *target method*, the method in that class, and a *target line*, the line in that method where the exception has been propagated. Then, it seeks to generate a unit test which replicates the given stack trace from the target frame (at level n) to the deepest frame (at level 1). For instance, if we pass the stack trace in Table 3.1 as the given trace and indicate the second frame as the target frame (level 2), the output of EvoCrash will be a unit test for the class `TempFile` which replicates first two frames of the given stack trace with the same type of the exception (`NullPointerException`).

3.2.2.1 Guided genetic algorithm

The search process in EvoCrash begins by randomly generating unit tests for the target frame. In this phase, called *guided initialization*, the target method corresponding to the selected frame (i.e., the *failing method* to which the exception is propagated) is injected in every randomly generated unit test. During subsequent phases of the search, *guided crossover* and *guided mutation*, standard evolutionary operations are applied to the unit tests. However, applying these operations involves the risk of losing the injected failing method. Therefore, the algorithm ensures that only unit tests with the injected failing method call remain in the evolution loop. If the generated test by crossover does not contain the failing method, the algorithm replaces it with one of its parents. Also, if after a mutation, the resulting test does not contain the failing method, the algorithm redoes the mutation until the the failing method is added to the test again. The search process continues until either the search budget is over or a crash reproducing test case is found.

To evaluate the generated tests, EvoCrash applies the following weighted sum fitness

function [204] to a generated test t:

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_{except}) + max(d_{trace}) & \textit{if the line is not reached} \\ 3 \times min(d_s) + 2 \times d_{except}(t) + max(d_{trace}) & \textit{if the line is reached} \\ 3 \times min(d_s) + 2 \times min(d_{except}) + d_{trace}(t) & \textit{if the exception is thrown} \end{cases} \quad (3.1)$$

Where:

- $d_s \in [0, 1]$ indicates the distance between the execution of t and the target statement s located at the target line. This distance is computed using the *approach level*, measuring the minimum number of control dependencies between the path of the code executed by t and s, and normalized *branch distance*, scoring how close t is to satisfying the branch condition for the branch on which$s$ is directly control dependent [160]. If the target line is reached by the test case, $d_l(t)$ equals to 0.0;

- $d_{except}(t) \in \{0, 1\}$ indicates if the target exception is thrown ($d_e = 0$) or not ($d_e = 1$);

- $d_{trace}(t) \in [0, 1]$ indicates the similarity of the input stack trace and the one generated by t by looking at class names, methods names and line numbers;

- $max(\cdot)$ denotes the maximum possible value for the function.

Since the stack trace similarity is relevant only if the expected exception is thrown by t, and the check whether the expected exception is thrown or not is relevant only if the target line where the exception propagates is reached, d_{except} and d_{trace} are computed only upon the satisfaction of two *constraints*: the target exception has to be thrown in the target line s and the stack trace similarity should be computed only if the target exception is actually thrown.

Unlike other stack trace similarity measures (e.g., [194]), Soltani *et al.* [204] do not require two stack traces to share the same common prefix to avoid rejecting stack traces where the difference is only in one intermediate frame. Instead, for each frame, $d_{trace}(t)$ looks at the closest frame and compute a distance value. Formally, for an original stack trace $S*$ and a test case t producing a stack trace S, $d_{trace}(t)$ is defined as follows:

$$d_{trace}(t) = \varphi\left(\sum_{f* \in S*} min\{diff(f*, f) : f \in S\}\right) \quad (3.2)$$

Where $\varphi(x) = x/(x+1)$ is a normalization function [160] and $diff(f*, f)$ measures

the difference between two frames as follows:

$$diff(f*,f) = \begin{cases} 3 & \textit{if the classes are different} \\ 2 & \textit{if the classes are equal but the methods are different} \\ \varphi(|l* - l|) & \textit{otherwise} \end{cases} \tag{3.3}$$

Where l (resp. $l*$) is the line number of the frame f (resp. $f*$).

Each of the three components if the fitness function defined in Equation 3.1 ranges from 0.0 to 1.0, the overall fitness value for a given test case ranges from 0.0 (crash is fully reproduced) to 6.0 (no test was generated), depending on the conditions it satisfies.

3.2.2.2 Comparison with the state-of-the-art

Crash reproduction tools. Table 3.2 presents the number of crashes used in the benchmarks used to evaluated stack-trace based post-failure crash reproduction tools as well as their crash reproduction rates. EvoCrash has been evaluated on various crashes reported in other studies and has the highest reproduction rate.

EvoSuite. Table 3.2 also reports the comparison of EvoCrash with EvoSuite, using exception coverage as the primary objective, applied by Soltani *et al.* [204]. All the crashes reproduced by EvoSuite could also be reproduced by EvoCrash on average 170% faster and with a higher reproduction rate.

3.3 Benchmark design

Benchmarking is a common practice to assess a new technique and compare it to the state of the art [199]. For instance, SF110 [105] is a sample of 100 Java projects from SourceForge, and 10 popular Java projects from GitHub, that may be used to assess (search based) test case selection techniques. In the same way, Defects4J [135] is a collection of bugs coming from popular open-source projects: for each bug, a buggy and a fixed version of the projects, as well as bug revealing test case, are provided. Defects4J is aimed to assess various testing techniques like test case selection or fault localization.

In their previous work, Soltani et al. [202], Xuan et al. [215], and Chen and Kim [81] used Apache Commons Collections [47], Apache Ant [46], and Apache Log4j [48] libraries. In addition to Apache Ant and Apache Log4j, Nayrolles et al. [173] used bug reports from 8 other open-source software.

Table 3.2: The number of crashes used in each crash reproduction tool experiment, the gained reproduction by them, and the involved projects.

Tool	Reproduced/Total	Rate	Projects
EvoCrash [202, 204]	46/54	85%	Apache Commons Collections Apache Ant Apache Log4j ActiveMQ DnsJava JFreeChart
EvoSuite [204]	18/54	33%	Apache Commons Collections Apache Ant Apache Log4j ActiveMQ DnsJava JFreeChart
STAR [81]	30/51	59%	Apache Commons Collections Apache Ant Apache Log4j
MuCrash [215]	8/12	66%	Apache Commons Collections
JCharming [173]	8/12	66%	Apache Ant Apache Log4j ActiveMQ DnsJava JFreeChart

In this paper we enhance previous efforts to build a benchmark dedicated to crash reproduction by collecting cases coming from both state of the art literature and actively maintained industrial open-source projects with well documented bug trackers.

3.3.1 Projects selection protocol

As Table 3.2 clearly shows, current crash reproduction tools are not evaluated using a common benchmark. This hampers progress in the field as it makes it hard to compare approaches. To be able to perform analysis of the results of a crash reproduction attempt, we define the following *benchmark requirements* for our benchmark:

BR1, to be part of the benchmark, the projects should have openly accessible binaries, source code, and crash stack traces (in an issue tracker for instance);

BR2, they should be under active maintenance to be representative of current software engineering practices and ease communication with developers;

BR3, each stack trace should indicate the version of the project that generated the stack trace; and

BR4, the benchmark should include projects of varying size.

To best of our knowledge, there is no benchmark fulfilling those requirements. The closest benchmark is Defects4j. However, only 25% of the defects manifest trough a crash stack trace (**BR1**) and the projects are relatively small (**BR4**). To address those limitations, we built a new benchmark dedicated to the crash reproduction tools.

To build our benchmark, we took the following approach. First, we investigated projects collected in SF110 [105] and Defects4J [135] as state of the art benchmarks. However, as most projects in SF110 have not been updated since 2010 or earlier, we discarded them from our analysis (**BR2**). From Defects4J, we collected 73 cases where bugs correspond to actual crashes: i.e., the execution of the test case highlighting the bug in a given buggy version of a project generates a stack trace that is not a test case assertion failure.

As also discussed by Fraser and Arcuri [105], to increase the representativeness of a benchmark, it is important to include projects that are popular and attractive to end-users. Additionally to Defects4J, we selected two industrial open-source projects: XWiki [216] and Elasticsearch [97]. XWiki is a popular enterprise wiki management system. Elasticsearch, a distributed RESTful search and analytic engine, is one of the ten most popular projects on GitHub[5]. To identify the top ten popular projects from Github, we took the following approach: (i) we queried the top ten projects that had the highest number of forks; (ii) we queried the top ten projects that had the highest number of stars; (iii) we queried the top ten trending projects; and (iv) took the intersection of the three.

Four projects were shared among the above top-ten projects, namely: Java-design-patterns [128], Dubbo [95], RxJava [195], and Elasticsearch. To narrow down the scope of the study, we selected Elasticsearch, which ranked the highest among the four shared projects.

[5]This selection was performed on 26/10/2017.

3.3.2 Stack trace collection and preprocessing

For each project, we collected stack traces to be reproduced as well as the project binaries, with specific versions on which the exceptions happened.

3.3.2.0.1 Defects4J. From the 395 buggy versions of the Defects4J projects, we kept only the bugs relevant to our crash reproduction context (73 cases), i.e., the bugs that manifest as crashes. We manually inspected the stack traces generated by the failing tests and collected those which are not JUnit assertion failures (i.e., those which are due to an exception thrown by the code under test and not by the JUnit framework). For instance, for one stack trace from the Joda-Time project:

```
0 java.lang.IllegalArgumentException:
1   at org.joda.time.Partial.<init>(Partial.java:224)
2   at org.joda.time.Partial.with(Partial.java:466)
3   at org.joda.time.TestPartial_Basics.testWith_baseAndArgHaveNoRange(...)
```

We only consider the first and second frames (lines 1 and 2). The third and following lines concern testing classes of the project, which are irrelevant for crash reproduction. They are removed from the benchmark, resulting in the following stack trace with two frames:

```
0 java.lang.IllegalArgumentException:
1   at org.joda.time.Partial.<init>(Partial.java:224)
2   at org.joda.time.Partial.with(Partial.java:466)
```

We proceeded in the same way for each Defects4J project and collected a total of 73 stack traces coming from five (out of the six) projects: JFreeChart, Commons-lang, Commons-math, Mockito, and Joda-Time. All the stack traces generated by the Closure compiler test cases are JUnit assertion failures.

3.3.2.0.2 Elasticsearch. Crashes for Elasticsearch are publicly reported to the issue tracker of the project on GitHub[6]. Therefore, we queried the reported crashes, which were labelled as bugs, using the following string `"exception is:issue label:bug"`. From the resulting issues (`600` approx.), we manually collected the most recent ones (reported since 2016), which addressed the following: (i) the version which crashed was reported, (ii) the issue was discussed by the developers and approved as a valid crash to be fixed. The above manual process resulted in 76 crash stack traces.

[6]`https://github.com/elastic/elasticsearch/issues`

3.3.2.0.3 XWiki. XWiki is an open source project which has a public issue tracker[7]. We investigated first 1000 issues which are reported for XWIK-7.2 (released in September 2015) to XWIK-9.6 (released in July 2017). We selected the issues where: (i) the stack trace of the crash was included in the reported issue, and (ii) the reported issue was approved by developers as a valid crash to be fixed. Eventually, we selected a total of 51 crashes for XWIKI.

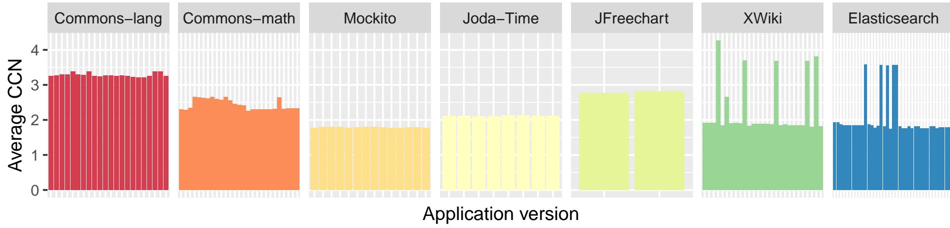
3.4 The JCrashPack benchmark

The result of our selection protocol is a benchmark with 200 stack traces called *JCrashPack*. For each stack trace, based on the information from the issue tracker and the Defects4J data, we collected: the *Java project* in which the crash happened, the *version* of the project where the crash happened and (when available) the *fixed version* or the fixing commit reference of the project; the *buggy frame* (i.e., the frame in the stack trace targeting the method where the bug lays); and the *Cyclomatic Complexity Number (CCN)* and the *Non-Commenting Sources Statements (NCSS)* of the project, presented in Figure 3.1. Due to the manual effort involved in filtering, verifying and cleaning up stack traces, issues, the collection of stack traces and binaries (including the project's dependencies binaries) took about 4.5 person-months in total.

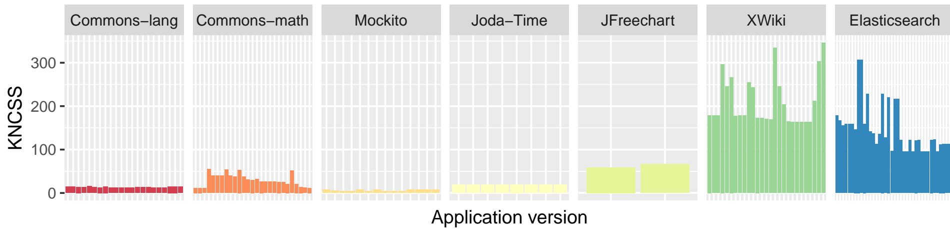
Figure 3.1 presents the average Cyclomatic Complexity Number (CCN) per method for each project and the Non-Commenting Sources Statements (NCSS) per project, ordered by version number, to give an idea of the complexity of a project. Also, Table 3.3 gives the number of versions and the average number of non-commenting source statement for each project in JCrashPack. As illustrated in the table and figure, JCrashPack contains projects of diverse complexities (the CCN for the least complex project is 1.77, and for the most complex is 3.38) and sizes (the largest project has 177,840 statements, and the smallest one holds 6,060 statements on average), distributed among different versions.

Table 3.4 shows the distribution of stack traces per exception type for the six most common ones, the *Other* category denoting remaining exception types. According to this table, the included stack traces in JCrashPack covers different types of the exceptions. Also, they are varied in the size (number of frames): the smallest stack traces have one frame and the largest, a user-defined exception in *Other*, has 175 frames.

[7] https://jira.xwiki.org/browse/XWIKI/



(a) Average methods Cyclomatic Complexity Number (CCN)



(b) Thousands of Non-Commenting Sources Statements (KNCSS)

Figure 3.1: Complexity and size of the different projects

JCrashPack is extensible and publicly available on GitHub.[8] We provide guidelines to add new crashes to the benchmark and make a pull request to include them in JCrashPack master branch. The detailed numbers for each stack trace and its project are available on the JCrashPack website.

3.5 Running experiments with ExRunner

We combine JCrashPack with ExRunner, a tool that can be used for running experiments with a given stack trace-based crash reproduction tool. This tool (i) facilitates the automatic parallel execution of the crash reproduction instances, (ii) ensures robustness in the presence of failures during the crash reproduction failure, and (iii) allows to plug different crash reproduction tools to allow a comparison of their capabilities.

Figure 3.2 gives an overview of ExRunner architecture. The *job generator* takes as input the stack traces to reproduce, the path to the Jar files associated to each stack trace, and the configurations to use for the stack trace reproduction tool under study. For each stack trace, the job generator analyzes the stack frames and discards those

[8] At `https://github.com/STAMP-project/JCrashPack`

Table 3.3: The number of versions and average number of statements ($\overline{NCSS}$) for each project.

Applications	Number of versions	$\overline{NCSS}$
Commons-lang	22	13.38k
Commons-math	27	29.98k
Mockito	14	6.06k
Joda-Time	8	19.41k
JFreechart	2	63.01k
XWiki	32	177.84k
Elasticsearch	46	124.36k
Total	151	62.01k

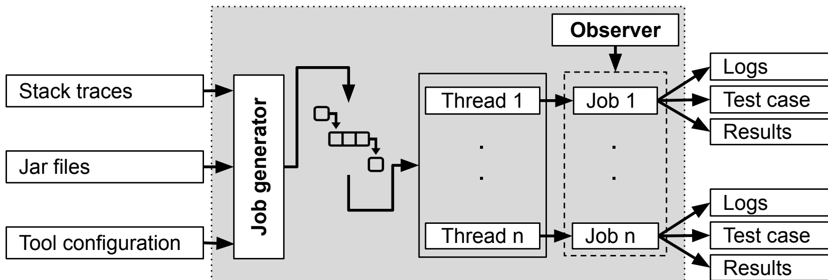


Figure 3.2: ExRunner overview

with a target method that does not belong to the target system, based on the package name. For instance, frames with a target method belonging to the Java SDK or other external dependencies are discarded from the evaluation. For each configuration and stack trace, the job generator creates a new job description (i.e., a JSON object with all the information needed to run the tool under study) and adds it to a queue.

To speed-up the evaluation, ExRunner multithreads the execution of the jobs. The number of threads is provided by the user in the configuration of ExRunner and depends on the resources available on the machine and required by one job execution. Each thread picks a job from the waiting queue and executes it. ExRunner users may activate an observer that monitors the jobs and takes care of killing (and reporting) those that do not show any sign of activity (by monitoring the job outputs) for a user-defined amount of time. The outputs of every job are written to separate files, with the generated test case (if any) and the results of the job execution (output results from the tool under study).

For instance, when used with EvoCrash, the log files contain data about the target

method, progress of the fitness function value during the execution, and branches covered by the execution of the current test case (in order to see if the line where the exception is thrown is reached). In addition, the results contain information about the progress of search (best fitness function, best line coverage, and if the target exception is thrown), and number of fitness evaluations performed by EvoCrash in an output CSV file. If EvoCrash succeeds to replicate the crash, the generated test is stored separately.

As mentioned by Fraser et al. [102], any research tool developed to generate test cases may face specific challenges. One of these is long (or infinite) execution time of the test during the generation process. To manage this problem, EvoSuite uses a timeout for each test execution, but sometimes it fails to kill sub-processes spawned during the search [102]. We also experienced EvoCrash freezing during our evaluation. In order to handle this problem, ExRunner creates an observer to check the status of each thread executing an EvoCrash instance. If one EvoCrash execution does not respond for 10 minutes (66% of the expected execution time), the Python script kills the EvoCrash process and all of its spawned threads.

Another challenge relates to garbage collection: we noticed that, at some point of the execution, one job (i.e., one JVM instance) allocated all the CPU cores for the execution of the garbage collector, preventing other jobs to run normally. Moreover, since EvoCrash allocates a large amount of heap space to each sub-process responsible to generate a new test case (since the execution of the target application may require a large amount of memory) [102], the garbage collection process could not retrieve enough memory and got stuck, stopping all jobs on the machine. To prevent this behaviour, we set *-XX:ParallelGCThreads* JVM parameter to 1, enabling only one thread for garbage collection, and limited the number of parallel threads per machine, depending on the maximal amount of allocated memory space. We set the number of active threads to 5 for running on virtual machines, and 25 for running on two powerful machines. Using the logging mechanism in EvoCrash, we are able to see when individual executions ran out of memory.

ExRunner is available together with JCrashPack.[9] It presently has only been used to perform EvoCrash benchmarking, yet it has been designed to be extensible to other available stack trace reproduction tools using a plugin mechanism. Integrating another crash reproduction tool requires the definition of two handlers, called by ExRunner: one to run the tool with the inputs provided by ExRunner (i.e. the stack trace, the target frame, and the classpath of the software under test); and one to parse the output produced by the tool to pick up relevant data (e.g., the final status

[9] See `https://github.com/STAMP-project/ExRunner`.

of the crash reproduction, progress of the tool during the execution, *etc.*). Relevant data are stored in a CSV file, readily available for analysis.[10]

3.6 Application to EvoCrash: setup

Having JCrashPack available allowed us to perform an extensive evaluation of EvoCrash, a state-of-the-art tool in search-based crash replication [204]. Naturally, our first research question deals with the capability of EvoCrash to reproduce crashes from JCrashPack:

RQ$_{1.1}$ *To what extent can EvoCrash reproduce crashes from JCrashPack?*

Since the primary goal of our evaluation is to identify current limitations, we refine the previous research question to examine which frames of the different crashes EvoCrash is able to reproduce:

RQ$_{1.2}$ *To what extent can EvoCrash reproduce the different frames of the crashes from JCrashPack?*

The diversity of crashes in JCrashPack also allows us to investigate how certain types of crashes affect reproducibility. Thus, we investigate whether the exception type and the project nature have an influence on the reproduction rate:

RQ$_{2.1}$ *How does project type influence performance of EvoCrash for crash reproduction?*

In addition, different types of projects might have impact on how costly it is to reproduce the reported crashes for them. The second research question studies the influence of the exception and project type on the performance of EvoCrash:

RQ$_{2.2}$ *How does exception type influence performance of EvoCrash for crash reproduction?*

Finally, we seek to understand why crashes could *not* be reproduced:

RQ$_{3}$ *What are the main challenges that impede successful search-based crash reproduction?*

[10] The ExRunner documentation includes a detailed tutorial describing how to proceed, available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application#run-other-crash-replication-tools-with-exrunner`.

3.6.1 Evaluation setup

Number of executions. Due to the randomness of Guided Genetic Algorithm in EvoCrash, we executed the tool multiple times on each frame. The number of executions has to strike a balance between the threats to external validity (i.e., the number of stack traces considered) and the statistical power (i.e., number of runs) [50, 105]. In our case, we do not compare EvoCrash to other tools (see for instance Soltani *et al.* [202, 204]), but rather seek to identify challenges for crash reproduction. Hence we favor external validity by considering a larger amount of crashes compared to previous studies [204] and ran EvoCrash 10 times on each frame. In total, we executed 18,590 EvoCrash runs.

Search parameters. We used the default parameter values [51, 105] with the following additional configuration options: we chose to keep the reflection mechanisms, used to call private methods, deactivated. The rationale behind this decision is that using reflection can lead to generating invalid objects that break the class invariant [154] during the search, which results in test cases helplessly trying to reproduce a given crash [81].

After a few trials, we also decided to activate the implementation of functional mocking available from EvoSuite [56] in order to minimize possible risks of environmental interactions on crash reproduction. Functional mocking works as follows: when, in a test case, a statement that requires new specific objects to be created (as parameters of a method call for instance) is inserted, either a plain object is instantiated by invoking its constructor, or (with a defined probability, left to its default value in our case) a mock object is created. This mock object is then refined using `when-thenReturn` statements, based on the methods called during the execution of the generated test case. Functional mocking is particularly useful in the cases where the required object cannot be successfully initialized (for instance, if it relies on environmental interactions or if the constructor is accessible only trough a factory).

Investigating the impact of those parameters and other parameters (e.g., crossover rate, mutation rate, etc. to overcome the challenges as identified in **RQ**3) is part of our future work.

Search budget. Since our evaluation is executed in parallel on different machines, we choose to express the budget time in terms of number of fitness evaluations: i.e., the number of times the fitness function is called to evaluate a generated test case during the execution of the guided generic algorithm. We set this number to 62,328, which corresponds to the average number of fitness evaluations performed by EvoCrash when running it during 15 minutes on each frame of a subset of 4 randomly selected

stack traces on one out of our two machines. Both of the machines have the same configuration: A cluster running Linux Ubuntu 14.04.4 LTS with 20 CPU-cores, 384 GB memory, and a 482 GB hard drive.

We partitioned the evaluation into two, one per available machine: all the stack traces with the same kind of exception have been run on one machine for 10 rounds. For each run, we measure the number of fitness evaluations needed to achieve reproduction (or the exhaustion of the budget if EvoCrash fails to reproduce the crash) and the best fitness value achieved by EvoCrash (0 if the crash is reproduced and higher otherwise). The whole process is managed using ExRunner. The evaluation itself was executed during 10 days on our 2 machines.

3.7 Application to EvoCrash: results

In this section, we answer the first two research questions on the extent to which the selected crashes and their frames were reproduced and the impact of the project and the exception type on the performance of EvoCrash. We detail the results by analyzing the outcome of EvoCrash in a majority of 10 executions for each frame of each stack trace. We classify the outcome of each execution in one of the five following categories:

reproduced: when EvoCrash generated a test that successfully reproduced the stack trace at the given frame level;

ex. thrown: when EvoCrash generated a test that cannot fully reproduce the stack trace, but covers the target line and throws the desired exception. The frames of the exception thrown, however, do not contain all the original frames;

line reached: when EvoCrash generated a test that covers the target line, but does not throw the desired exception;

line not reached: when none of the tests produced by EvoCrash could cover the target line within the available time budget; and

aborted: when EvoCrash could not generate an initial population to start the search process.

Each outcome denotes a particular state of the search process. For the *reproduced* frames, EvoCrash could generate a crash-reproducing test within the given time budget (here, 62,328 fitness evaluations). For the frames that could not be reproduced, either EvoCrash exhausted the time budget (for *ex. thrown*, *line reached*, and *line not reached* outcomes) or could not perform the guided initialization (i.e., generate at least one

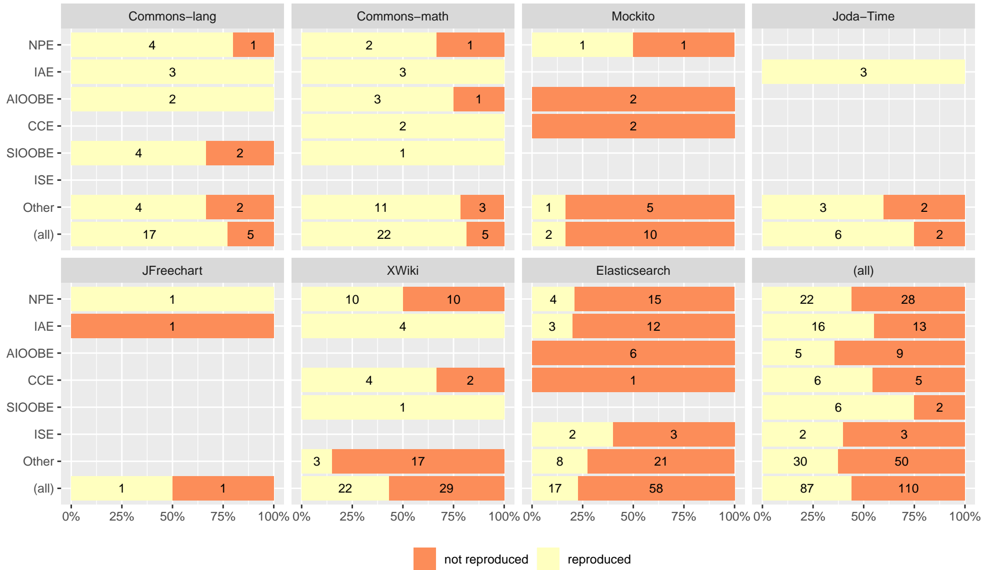


Figure 3.3: Reproduction outcome for the different crashes

test case with the target method) and did not start the search process (*aborted* outcomes). For instance, if the class in the target frame is abstract, EvoCrash may fail to find an adequate implementation of the abstract class to instantiate an object of this class during the guided initialization.

3.7.1 Crash Reproduction Outcomes (RQ1)

For **RQ**$_1$, we first look at the reproduced and non-reproduced crashes to answer **RQ**$_{1.1}$. If EvoCrash was successful in reproducing any frame of a stack trace in a majority of 10 executions, we count the crash as a **reproduced crash**. Otherwise, we count the crash as **not reproduced**. To answer **RQ**$_{1.2}$, we detail the results by analyzing the outcome of EvoCrash in a majority of 10 executions for each frame of each stack trace.

Figure 3.3 shows the number of reproduced and not reproduced crashes for each project (and all the projects) and type of exception. EvoCrash is successful in reproducing the majority of crashes (more than 75%) from *Commons-lang*, *Commons-math*, and *Joda-Time*. For the other projects, EvoCrash reproduced 50% or less of the crashes, with only 2 out of 12 crashes reproduced for *Mockito*. Crashes with an *IllegalArgumentException* are the most frequently reproduced crashed: 16 out of 29 (55%).

Before detailing the results of each frame of each crash, we first look at the frame levels that could be reproduced. Figure 3.4 presents for the 87 stack traces that could be reproduced, the distribution of the highest frame level that could be reproduced for the different crashes for each type of exception (in Figure 3.4a) and each application (in Figure 3.4b). As we can see, EvoCrash replicates lower frame levels more often than higher levels. For instance, for 39 out of the 87 reproduced stack traces, EvoCrash could not reproduce frames beyond level 1 and could reproduce frames up to level 5 for only 9 crashes.

Figure 3.4a indicates that EvoCrash can replicate only the first frame in 14 out of 22 NPE crashes, while there is only one NPE crash for which EvoCrash could reproduce a frame above level 3. In contrast, it is more frequent for EvoCrash to reproduce higher frame levels of IAE stack traces: the highest reproduced frames in 6 out of 16 IAE crashes are higher than 3. Those results suggest that, when trying to reproduce a crash, propagating an illegal argument value trough a chain of method calls (i.e., the frames of the stack trace) is easier than propagating a `null` value. According to Figure 3.4b, EvoCrash can reproduce frames higher than 6 only for *Commons-math* crashes. The highest reproduced frames in most of the reproduced crashes in this project are higher than level 2 (12 out of 22). In contrast, for *Elasticsearch* the highest reproduced frame is 1 in most of the crashes.

Both the number of crashes reproduced and the highest level at which crashes could be reproduced confirm the relevance of our choice to consider crashes from XWiki and Elasticsearch, for which the average number of frames (resp. 27.5 and 17.7) is higher than for Defects4J projects (at most 6.0 for JFreeChart), as they represent an opportunity to evaluate and understand current limitations.

3.7.1.1 Frames Reproduction Outcomes

To answer $\mathbf{RQ}_{1.2}$, we analyze the results for each frame individually. Figure 3.5 presents a summary of the results with the number of frames for the different outcomes. Figure 3.6 details the same results by application and exception.

Overall, we see in Figure 3.5 that EvoCrash reproduced 171 frames (out of 1,859), from 87 different crashes (out of 200) in the majority of the ten rounds. If we consider the frames for which EvoCrash generated a crash-reproducing test at least once in the ten rounds, the number of reproduced frames increases to 201 (from 96 different crashes). In total, EvoCrash exhausted the time budget for 950 frames: 219 with a test case able to throw the target exception, 245 with a test case able to reach the target line, and 486 without a test case able to reach the line. EvoCrash aborted

the search for 738 frames, 455 of which were from Elasticsearch, the application for which EvoCrash had the most difficulties to reproduce a stack trace.

Figure 3.6 details the results by applications (columns) and exceptions (lines). The last line (resp. column), denoted *(all)*, provides the global results for the applications (resp. exceptions). In the remainder of this section, we discuss the results for the different applications and exceptions.

3.7.1.2 Defects4J applications

For the Defects4J applications, presented in the first five columns in Figure 3.6, in total, 90 (out of 244) of the frames from 48 (out of 71) different crashes were *reproduced*. For 94 frames, EvoCrash exhausted the time budget (46 *ex. thrown*, 25 *line reached*, and 23 *line not reached*) and *aborted* for 60 frames from the Defects4J projects.

In particular, only 4 frames out of 61 frames for Mockito were successfully reproduced. For instance, EvoCrash could not reproduce `MOCKITO-4b`, which has only one frame. From our evaluation, we observe that one very common problem when trying to reproduce a *ClassCastException* is to find which class should be used to trigger the exception.

```
public void noMoreInteractionsWantedInOrder(Invocation undesired){
  throw new VerificationInOrderFailure(join( ...,
      " ... " + undesired.getMock() + "':", ...) );
}
```

The exception happens when the `undesired.getMock()` call returns an object that cannot be cast to `String`. During the search, EvoCrash mocks the `undesired` object and assigns some random value to return when the `getMock` method is called. EvoCrash generates a test able to cover the target line, but failing to trigger an exception. Since the signature of this method is `Object getMock()`, EvoCrash assigns only random `Object` values to return, where, from the original stack trace, a `Boolean` value is required to trigger the exception.

3.7.1.3 XWiki and Elasticsearch

XWiki is one of the industrial open source cases in the evaluation, for which 53 (out of 706) frames were successfully *reproduced*, 430 could not be reproduced with the

given time budget (125 *ex. thrown*, 127 *line reached*, and 178 *line not reached*), and 223 *aborted* during the generation of the initial population. EvoCrash *reproduced* only 28 (out of 909) frames from Elasticsearch, for which, the majority of frames (455) *aborted* during the generation of the initial population. However, EvoCrash was able to start the search for 426 frames (48 *ex. thrown*, 93 *line reached*, and 285 *line not reached*).

3.7.1.3.1 Variability of the reproductions. We also observed that XWiki and Elasticsearch have the highest variability in their outcomes. For XWiki (resp. Elasticsearch), 4 (resp. 3) frames that could be reproduced in a majority of time could however not be reproduced 10 out of 10 times, compared to 2 frames for Commons-lang and Commons-math. This could indicate a lack of guidance in the current fitness function of EvoCrash. For instance, for the Elasticsearch crash ES-26833, EvoCrash could only reproduce the third frame 4 times out of 10 and was therefore not considered as reproduced. After a manual inspection, we observed that EvoCrash gets stuck after reaching the target line and throwing the expected exception. From the intermediate test cases generated during the search, we see that the exception is not thrown by the target line, but a few lines after. Since the fitness value improved, EvoCrash got stuck into a local optima, hence the lower frequency of reproduction for that frame.[11] Out future work includes improvement of the guidance in the fitness function and a full investigation of the fitness landscape to decrease the variability of EvoCrash outcomes.

3.7.1.3.2 Importance of large industrial applications. Compared to Defects4J and XWiki applications, the crash reproduction rate drops from 36.9% for Defects4J, to 7.5% for XWiki, and only 3% for Elasticsearch. Those results emphasize the importance of large industrial applications for the assessment of search-based crash reproduction and enforce the need of context-driven software engineering research to identify relevant challenges [72].

Additionally to the larger variability of reproduction rate, we observe that frequent use of *Java generics* and *static initialization*, and most commonly, automatically generating suitable input data that resembles `http` requests are among the major reasons for the encountered challenges for reproducing Elasticsearch crashes. In Section 3.8 we will describe **14** categories of challenges that we identified as the underlying causes for the presented execution outcomes.

[11] A detailed analysis is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Elasticsearch/ES-26833.md`

3.7.1.4 Exceptions

The lines in Figure 3.6 presents the outcomes for the different exceptions. In particular, NPE, IAE, AIOOBE, and CCE are the most represented exceptions in JCrashPack. For those exceptions, EvoCrash could reproduce, respectively, 32 (out of 499), 40 (out of 250), 6 (out of 99), and 10 (out of 72) frames. Looking at the reproduction frequency, IAE is the most frequently reproduced exception (16%), followed by CCE (13.8%), NPE (6.4%), and AIOOBE (6%).

This contrast with the number of frames for which EvoCrash aborted the search, where NPE has the lowest frequency (181 frames, 36.2%), followed by IAE (101 frames, 40.4%), CCE (30 frames, 41.6%), and AIOOBE (48 frames, 48.4%). Interestingly, those numbers show that EvoCrash is able to complete the guided initialization for NPEs more often than for other exceptions.

Figure 3.6 also shows that the number of test cases that reach the line is low for NPEs, meaning that whenever EvoCrash generates at test able to cover the line (*line reached*), the evolution process will be able to progress and generate another test that throws an exception (*ex. thrown*).

Summary (RQ$_1$) *To what extent can EvoCrash reproduce crashes from JCrashPack, and how far it can proceed in the stack traces?* Overall, EvoCrash reproduced 171 frames (out of 1,859 - 9%), from 87 different crashes (out of 200 - 43.5%) in a majority out of 10 executions. Those numbers climb to 201 frames (10.8%) from 96 crashes (48%) if we consider at least one reproduction in one of the 10 executions. In most of the reproduced crashes, EvoCrash can only reproduce the first two frames. It indicates that since EvoCrash needs higher accuracy in setting the state of the software under test for reproducing higher frames, increasing the length of the stack trace reduces the chance of this tool for crash reproduction. When looking at larger industrial applications, the crash reproduction rates drop from 36.9% for Defects4J to 7.5% for XWiki and 3% for Elasticsearch. The most frequently reproduced exceptions are IllegalArgumentExceptions. The exceptions for which EvoCrash is the most frequently able to complete the guided initialization are NullPointerExceptions.

3.7.2 Impact of Exception Type and Project on Performance (RQ2)

To identify the distribution of fitness evaluations per exception type and project, we filtered the *reproduced* frames out of the 10 rounds of execution. Tables 3.5 and 3.6 present the statistics for these executions, grouped by application and exception types, respectively.

We filtered out the frames that were not reproduced to analyze the impact of project and exception types on the average number of fitness evaluations and, following recommendations by Arcuri and Briand [50], we replaced the test of statistical difference by a confidence interval. For both groups, we calculated confidence intervals with a 95% confidence level for medians with bootstrapping with $100,000$ runs.[12]

As Table 3.5 shows, for four projects (Commons-lang, Mockito, XWiki, and Elasticsearch) the median number of fitness evaluations is low. On the contrary, the cost of crash reproductions for Commons-math, Joda-Time, and JFreechart are higher in comparison to the rest of projects. By comparing those results with the projects sizes reported in Table 3.3, where the largest projects are XWiki (with $\overline{NCSS} = 177.84k$) and Elasticsearch (with $\overline{NCSS} = 124.36k$), we observe that the effort required to reproduce a crash cannot be solely predicted by the project size. This is consistent with the intuition that the difficulty of reproducing a crash only depends on the methods involved in the stack trace.

Similarly, according to Figure **??**, the average CCN for Mockito, XWiki, and Elasticsearch is lower compared to other projects. Table 3.5 shows that reproducing crashes from these projects is less expensive, and that reproducing crashes from Commons-math, Joda-Time, and JFreechart, which all have higher average CCN, is more expensive. We also observe that the average CCN for Commons-lang is high, however, contradicting the intuition that crashes from projects higher CCN are more expensive to reproduce, the cost for reproducing crashes in Commons-lang is low compared to other projects. This can be explained by the levels of the frames reproduced by EvoCrash: according to Figure 3.4, the average level of the reproduced frames in the crashes from Commons-lang is low compared to the other projects and, as we discussed in the previous section, reproducing crashes with fewer frames is easier for EvoCrash.

In general, we observe that the performance of EvoCrash depends on the complexity of the project and the frame level in the stack trace. Future work includes further investigations to determine which other factors (e.g., code quality) can influence EvoCrash performance.

From Table 3.6, we observe that for *CCE*, *SIOOBE*, and *AIOOBE*, the cost of generating a crash-reproducing test case is high, while for *NPE*, *IAE*, and *ISE*, the cost is lower. One possible explanation could be that generating input data which is in a suitable state for causing cast conflicts, or an array which is in the right state to be accessed

[12] We used the *boot* function from the *boot* library in R to compute the *basic* intervals with bootstrapping. See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results` to reproduce the statistical analysis.

by an illegal index is often non-trivial.

In contrast, to trigger an NPE, it is often enough to return a `null` value not checked by the crashing method. For example, Listing 3.1 shows the stack trace of CHART-4b, a crash from the JFreeChart application. The crash happens at line 1490 of the `createScatterPlot` method presented in Listing 3.2. Listing 3.3 shows the test case generated by EvoCrash that reproduces the 6th frame (line 6 in Listing 3.1) of the stack trace. First, the test initializes the mocks used as mandatory parameters values (from line 2 to 4), before calling the `createScatterPlot` method (at line 5). The `ds` `XYDataset` mock is used along the various calls (from line 6 to 1 in Listing 3.1), up to the method `getDataRange` presented in Listing 3.4 that triggers the NPE at line 4493. In our case, the `null` value is returned by the `getRendererForDataset` call with the propagated `ds` mock at line 4491.

Listing 3.1: Stack trace for the crash CHART-4b

```
0  java.lang.NullPointerException
1      at org.jfree.chart.plot.XYPlot.getDataRange(XYPlot.java:4493)
2      at org.jfree.chart.axis.NumberAxis.autoAdjustRange(NumberAxis.java:434)
3      at org.jfree.chart.axis.NumberAxis.configure(NumberAxis.java:417)
4      at org.jfree.chart.axis.Axis.setPlot(Axis.java:1044)
5      at org.jfree.chart.plot.XYPlot.<init>(XYPlot.java:660)
6      at org.jfree.chart.ChartFactory.createScatterPlot(ChartFactory.java:1490)
```

Listing 3.2: Code excerpt from JFreeChart `ChartFactory.java`

```
1478 public static JFreeChart createScatterPlot(String title, String xAxisLabel,
1479     String yAxisLabel, XYDataset dataset, PlotOrientation orientation,
1480     boolean legend, boolean tooltips, boolean urls) {
1481
1482   if (orientation == null) {
1483    throw new IllegalArgumentException("Null 'orientation' argument.");
1484   }
1485   NumberAxis xAxis = new NumberAxis(xAxisLabel);
1486   xAxis.setAutoRangeIncludesZero(false);
1487   NumberAxis yAxis = new NumberAxis(yAxisLabel);
1488   yAxis.setAutoRangeIncludesZero(false);
1489
1490   XYPlot plot = new XYPlot(dataset, xAxis, yAxis, null);
1491
1492    [...]
```

```
1493 }
```

Listing 3.3: The test case generated by EvoCrash for reproducing the 6th frame of CHART-4b

```
1 public void test() throws Throwable {
2  XYDataset ds = mock(XYDataset.class, new ViolatedAssumptionAnswer());
3  doReturn(0).when(ds).getSeriesCount();
4  PlotOrientation pl = mock(PlotOrientation.class, new
       ViolatedAssumptionAnswer());
5  ChartFactory.createScatterPlot((String) null, (String) null, (String) null, ds,
       pl, true, true, true);
6 }
```

Listing 3.4: Code excerpt from JFreeChart `XYPlot.java`

```
4490 public Range getDataRange(ValueAxis axis) {
4491  XYItemRenderer r = getRendererForDataset(d); // d == ds and
        getRendererForDataset(d) returns null
4492   [...]
4493  Collection c = r.getAnnotations(); // r is null and throws a NPE
4494   [...]
4495 }
```

Considering the presented results in Figure 3.6 and Table 3.5, crash replication for various exceptions may be dependent on project type. Figure 3.7 presents the results of crash reproduction grouped both by applications and exception types. As the figure shows, the cost of reproducing NPE is lower for Elasticsearch, compared to XWiki and JFreechart, and the cost of reproducing IAE is lower for Commons-lang than for Elasticsearch. We also observe differences in terms of costs of reproducing AIOOBE and SIOOBE for different projects.

Summary (RQ$_{2.1}$) *How does project type influence performance of EvoCrash for crash reproduction?*

We observed that the factors are (i) the complexity of the the project, and (ii) the level of the reproduced frames (reproducing higher frame requires more effort). Furthermore, we see no link between the size of the project and the effort required to reproduce one of its crashes.

Summary (RQ$_{2.2}$) How does exception type influence performance of EvoCrash for crash reproduction?

For the exceptions, we observe that for ClassCastException, ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException, the cost of generating a crash-reproducing test case is high, while for NullPointerException, IllegalArgumentException, and IllegalStateException, the cost is lower. This result indicates that the cost of reproducing types of exceptions for a non-trivial scenario (*e.g.*, class conflicts or accessing an illegal state of an array) needs a more complex input generation. Furthermore, accessing the corresponding complex state is more time consuming for the search process.

3.8 Challenges for crash reproduction (RQ3)

To identify open problems and future research directions, we manually analyzed the execution logs of 1,653 frames that could not be reproduced in any of the 10 executions. This analysis includes a description of the reason why a frame could not be reproduced.[13] Based on those descriptions, we grouped the reason of the different failures into 13 categories and identified future research directions. Table 3.7 provides the number and frequency of frames classified in each category.[14] The complete categorization table is available in our replication package.[15]

For each challenge, we discuss to what extent it is crash-reproduction-specific and its relation to search-based software testing in general. In particular, for challenges previously identified by the related literature in search-based test case generation, we highlight the differences originating from the crash reproduction context.

3.8.1 Input data generation

Generating complex input objects is a challenge faced by many automated test generation approaches, including search-based software testing and symbolic execution

[13] Available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results/manual-analysis`.

[14] For each category, we provide illustrative examples from `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results/examples`.

[15] The full table is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/categorisation.csv`.

Listing 3.5: Excerpt of the stack trace for the crash XWIKI-13708

```
0 java.lang.NullPointerException: null
1   at com.xpn.xwiki.internal.template.TemplateListener.onEvent(TemplateListener.
2       java:79)
3   at org.xwiki.observation.internal.DefaultObservationManager.notify(Default
4       ObservationManager.java:307)
5   at org.xwiki.observation.internal.DefaultObservationManager.notify(Default
6       ObservationManager.java:269)
7    [...]
```

[70]. Usually, the input space of each input is large and generating proper data enabling the search process to cover its goals is difficult.

As we can see from Table 3.7, this challenge is substantial in search-based crash reproduction. Trying to replicate a crash for a target frame requires to set the input arguments of the target method and all the other calls in the sequence properly such that when calling the target method, the crash happens. Since the input space of a method is usually large, this can be challenging. EvoCrash uses randomly generated *input arguments* and mock objects as inputs for the target method. As we described in Section 3.7, we observe that a widespread problem when reproducing a *ClassCast-Exception* (CCE) is to identify which types to use as input parameters such that a CCE is thrown. In the case of a CCE, this information can be obtained from the error message of the exception. Our future work includes harvesting additional information, like error messages, to help the search process.

We also noticed that some stack traces involving Java *generic types* make EvoCrash abort the search after failing to inject the target method in every generated test during the guided initialization phase. Generating *generic type* parameters is also a recognized challenge for automated testing tools for Java [104]. To handle these parameters, EvoCrash, based on EvoSuite's implementation [104], collects candidate types from `castclass` and `instanceof` operators in Java bytecode, and randomly assign them to the type parameter. Since the candidate types may themselves have generic type parameters, a threshold is used to avoid large recursive calls to generic types. One possible explanation for the crashes in these cases could be that the threshold is not correctly tuned for the kind of classes involved in the recruited projects. Thus, the tool fails to set up the target method to inject to the tests. Based on the results of our evaluation, handling Java generics in EvoCrash needs further investigation to identify the root cause(s) of the crashes and devise effective strategies to address them.

Listing 3.6: Code excerpt from method `onEvent` in `TemplateListener.java`

```
72 public void onEvent(Event event, Object source, Object data) {
73   XWikiDocument document = (XWikiDocument) source;
74
75   if (document.getXObject(WikiSkinUtils.SKINCLASS_REFERENCE) != null) {
76     if (event instanceof AbstractAttachmentEvent) {
77       XWikiAttachment attachment =
             document.getAttachment(((AbstractAttachmentEvent)
             event).getName());
78       String id = this.referenceSerializer.serialize(attachment.getReference());
             // target line
79       [...]
80     }
81   }
82 }
```

For instance, EvoCrash cannot reproduce the first frame of crash XWIKI-13708[16], presented in Listing 3.5. The target method `onEvent` (detailed in Listing 3.6) has three parameters. EvoCrash could not reach the target line (line 78 in Listing 3.6) as it failed to generate a fitted value for the second parameter (`source`). This (`Object`) parameter should be castable to `XWikiDocument` and should return values for `getXObject()` or `getAttachment()` (using mocking for instance).

Chosen examples: XWIKI-13708, frame 1; ES-22922, frame 5; ES-20479, frame 10.[17]

3.8.2 Complex code

Generating tests for complex methods is hard for any search-based software testing tool [119]. In this study, we indicate a method as complex if (i) it contains more than 100 lines of code and high cyclomatic complexity; (ii) it holds nested predicates [119, 158]; or (iii) it has the *flag problem* [158, 161], which include (at least one) branch predicate with a binary (boolean) value, making the landscape of the fitness function flat and turning the search into a random search [119].

[16] `https://jira.xwiki.org/browse/XWIKI-13708`

[17] See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/InputDataGeneration.md`.

Listing 3.7: Stack trace for the crash XWIKI-12584

```
0 java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to java.lang.String
1   at [...].XWikiHibernateStore.searchDocumentReferencesInternal([...]:2457)
2   at [...].XWikiHibernateStore.searchDocumentsNamesInternal([...]:2440)
3   at [...].XWikiHibernateStore.searchDocumentsNames([...]:2246)
4   at [...].XWikiHibernateStore.searchDocumentsNames([...]:2230)
5   at [...].XWikiCacheStore.searchDocumentsNames([...]:373)
6   at [...].XWiki.searchDocuments([...]:576)
```

As presented in Section 4.2, the first component of the fitness function that is used in EvoCrash encodes how close the algorithm is to reach the line where the exception is thrown. Therefore, frames of a given stack trace pointing to methods with a high code complexity[18] are more costly to reproduce, since reaching the target line is more difficult.

Handling complex methods in search-based crash reproduction is harder than in general search-based testing. The search process in crash reproduction should cover (in most cases) only one specific path in the software under test to achieve the reproduction. If there is a complex method on this path, the search process cannot achieve reproduction without covering it. Unlike the more general coverage driven search-based testing approach (with line coverage for instance), where the are usually multiple possible executions paths to cover a goal.

Chosen examples: XWIKI-13096, frame 3; ES-22373, frame 10.[19]

3.8.3 Environmental dependencies

As discussed by Arcuri et al. [54], generating unit tests for classes which interact with the environment leads to (i) difficulty in covering certain branches which depend on the state of the environment, and (ii) generating flaky tests [155], which may sometimes pass, and sometimes fail, depending on the state of the environment. Despite the numerous advances made by the search-based testing community in handling environmental dependencies [54, 105], we noticed that having such dependencies in the target class hampers the search process. Since EvoCrash builds on top of EvoSuite [103], which is a search-based *unit* test generation tool, we face the same problem in the crash reproduction problem as well.

[18]In some cases for Elasticsearch, the failing methods have nearly 300 lines of source code.
[19]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/ComplexCode.md`.

For instance, Listing 3.7 shows the stack trace of the crash XWIKI-12584.[20] During the evaluation, EvoCrash could not reproduce any of the frames of this stack trace. During our manual analysis, we discovered that, for the four first frames, EvoCrash was unable to instantiate an object of class `XWikiHibernateStore`,[21] resulting in an abortion of the search. Since the class `XWikiHibernateStore` relies on a connection to an environmental dependency (here, a database), generating unit test requires substantial mocking code[22] that is hard to generate for EvoCrash. As for input data generation, our future work includes harvesting and leveraging additional information from existing tests to identify and use relevant mocking strategies.

Chosen examples: ES-21061, frame 4; XWIKI-12584, frame 4.[23]

3.8.4 Static initialization

In Java, static initializers are invoked only once when the class containing them is loaded. As explained by Fraser and Arcuri [105], these blocks may depend on static fields from other classes on the classpath that have not been initialized yet, and cause exceptions such as `NullPointerException` to be thrown. In addition, they may involve environmental dependencies that are restricted by the security manager, which may also lead to unchecked exceptions being generated.

In our crash reproduction benchmark, we see that about 9% (see Table 3.7) of the cases cannot be reproduced as they point to classes that have static initializers. When such frames are used for crash reproduction with EvoCrash, the tool currently aborts the search without generating any crash reproducing test.

As Fraser and Arcuri [105] discuss, automatically determining and solving all possible kinds of dependencies in static initializers is a non-trivial task that warrants dedicated research.

Chosen examples: ES-20045, frames 1 and 2.[24]

[20] Reported at `https://jira.xwiki.org/browse/XWIKI-12584` and analyzed at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Xwiki/XWIKI-12584.md`.

[21] See `https://github.com/xwiki/xwiki-platform/blob/xwiki-platform-7.2-milestone-2/xwiki-platform-core/xwiki-platform-oldcore/src/main/java/com/xpn/xwiki/store/XWikiHibernateStore.java`

[22] See `https://github.com/xwiki/xwiki-platform/blob/xwiki-platform-7.2-milestone-2/xwiki-platform-core/xwiki-platform-oldcore/src/test/java/com/xpn/xwiki/store/XWikiHibernateStoreTest.java`

[23] See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/EnvironmentalDependencies.md`.

[24] See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/`

3.8.5 Abstract classes and methods

In Java, abstract classes cannot be instantiated. Although generating coverage driven unit tests for abstract classes is possible (one would most likely generate unit tests for concrete classes extending the abstract one or use a parametized test to check that all implementations respect the contract defined by the abstract class), when a class under test is abstract, EvoSuite (as the general test generation tool for java) looks for classes on the classpath that extend the abstract class to create object instances of that class. In order to cover (e.g., using line coverage) specific parts of the abstract class, EvoSuite needs to instantiate the right concrete class allowing to execute the different lines of the abstract class.

For crash reproduction, as we can see from Table 3.7, it is not uncommon to see abstract classes and methods in a stack trace. In several cases from Elasticsearch, the majority of the frames from a given stack trace point to an abstract class. Similarly to coverage-driven unit test generation, EvoCrash needs to instantiate the right concrete class: if EvoCrash picks the same class that has generated the stack trace in the first place, then it can generate a test for that class that reproduces the stack trace. However, if EvoCrash picks a different class, it could still generate a test case that satisfies the first two conditions of the fitness function (section 4.2). In this last case, the stack trace generated by the test would match the frames of the original stack trace, as the class names and line numbers would differ. The fitness function would yield a value between 0 and 1, but it may never be equal to 0.

Chosen examples: ES-22119, frames 3 and 4; XRENDERING-422, frame 6.[25]

3.8.6 Anonymous classes

As discussed in the study by Fraser *et al.* [103], generating automated tests for covering anonymous classes is more laborious because they are not directly accessible. We observed the same challenge during the manual analysis of crash reproduction results generated by EvoCrash. When the target frame from a given crash stack trace points to an anonymous object or a lambda expression, guided initialization in EvoCrash fails, and EvoCrash aborts the search without generating any test.

Chosen examples: ES-21457, frame 8; XWIKI-12855, frames 30 and 31.[26]

master/results/examples/StaticInitialisation.md.

[25]See https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/AbstractClass.md.

[26]See https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/AnonymousClass.md.

3.8.7 Private inner classes

Since it is not possible to access a private inner class, and therefore, not possible to directly instantiate it, it is difficult for any test generation tool in Java to create an object of this class. As for anonymous classes, this challenge is also present for crash reproduction approaches. In some crashes, the target frame points to a failing method inside a private inner class. Therefore, it is not possible to directly inject the failing method from this class during the guided initialization phase, and EvoCrash aborts the search.

Chosen example: MATH-58b, frame 3.[27]

3.8.8 Interfaces

In 6 cases, the target frame points to an interface. In Java, similar to abstract classes, interfaces may not be directly instantiated. In these cases also, EvoCrash randomly selects the classes on the classpath that implement the interface and, depending on the class picked by EvoCrash, the fitness function may not reach `0.0` during the search if the class is different from the one used when the input stack trace has been generated. This category is a special case of *Abstract classes and methods* (described in Section 3.8.5), however, since the definition of a default behavior for an interface is a feature introduced by Java 8 [175] that has, to the best of our knowledge, not been previously discussed for search-based testing, we choose to keep it as a separate category.

Chosen example: ES-21457, frame 9.[28]

3.8.9 Nested private calls

In multiple cases, the target frame points to a private method. As we mentioned in Section 3.6, those private methods are not directly accessible by EvoCrash. To reach them, EvoCrash detects other public or protected methods which invoke the target method directly or indirectly and randomly choose during the search. If the chain of method calls, from the public caller to the target method, is too long, the likelihood that EvoCrash may fail to pick the right method during the search increases.

[27] See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/PrivateInnerClass.md`.

[28] See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/Interface.md`.

In general, calling private methods is challenging for any automated test generation approach. For instance, Arcuri *et al.* [56] address this problem by using the Java reflection mechanism to access private methods and private attributes during the search. As mentioned in Section 3.6.1, this can generate invalid objects (with respect to their class invariants) and lead to generating test cases helplessly trying to reproduce a given crash [81].

Chosen examples: XRENDERING-422, frames 7 to 9.[29]

3.8.10 Empty `enum` type

In the stack trace of the ES-25849 crash,[30] the 4th frame points to an empty enumeration Java type.[31] Since there are no values in the enumeration, EvoCrash was not able to instantiate a value and *aborted* during the initialization of the population. Frames pointing to code in an empty enumeration Java type should not be selected as target frames and could be filtered out using a preliminary static analysis.

Chosen example: ES-25849, frame 4.

3.8.11 Frames with `try`/`catch`

Some frames have a line number that designates a call inside a `try/catch` block. When the exception is caught, it is no longer thrown at the specific line given in the trace, rather it is typically handled inside the associated `catch` blocks. From what we observed, often catch blocks either (i) re-throw a checked exception, which yield chained stack traces with information that is not exactly as the input stack trace but can still be used for crash reproduction; or (ii) log the caught exception. Since EvoCrash only considers uncaught exceptions that are generated as the result of running the generated test cases during the search, the logged stack traces is presently no use for crash reproduction. Also, even if a stack trace is recorded to an error log, this stack trace is not the manifestation of a crash *per se*. Indeed, once the exception logged, the execution of the program continues normally.

[29]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/NestedPrivateCalls.md`.
[30]The analysis is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Elasticsearch/ES-25849.md`.
[31]See `https://github.com/jimczi/elasticsearch/blob/0a4b38b60c2752cdc6de819f5\bf3414bd01f88c5/core/src/main/java/org/elasticsearch/index/fielddata/ordinals/GlobalOrdinalsBuilder.java`.

For instance, for the crash ES-20298,[32] EvoCrash cannot reproduce the fourth frame of the crash. This frame points to the following method call in a `try` and `catch`:

```
try {
    processResponse(response);
} catch (Throwable t) {
    onFailure(t);
}
```

Even if an exception is thrown by the `processResponse` method, this exception is caught and logged, and the execution of the program continues normally.

Generally, if an exception is caught in one frame, it cannot be reproduced (as it cannot be observed) from higher level frames. For instance, for ES-20298, all frames above level 4 cannot be reproduced since the exception is catch in frame 4 and not propagated to the higher frames. This property of a crash stack trace implies that, for now, depending on where in the trace such frames exist, only a fraction of the input stack traces can actually be used for automated crash reproduction. Future development of EvoCrash can alleviate this limitation by, additionally to the monitoring of uncaught exceptions, read the error log to affecting the propagation of exceptions during execution. However, unlike other branching instructions relying on boolean values, for which classical coverage driven unit test generation can use the *branch distance* (see Section 3.2.2.1) to guide the search [160], there is little guidance offered for `try/catch` instructions since the branching condition is implicit in one or more instructions in the `try`.

Chosen example: ES-14457, frame 4.[33]

3.8.12 Missing line number

31 frames in JCrashPack have frames with a missing line number, as shown in Listing 3.8. This happens if the Java files have been compiled without any *debug* information (by default, the Java compiler add information about the source files and line numbers, for instance, when printing a stack trace) or if the frame points to a class

[32] Reported at `https://github.com/elastic/elasticsearch/issues/20298` and analyzed at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Elasticsearch/ES-20298.md`

[33] See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/TryCatch.md`.

Listing 3.8: An excerpt of the stack trace from the crash XRENDERING-422 with missing line numbers

```
at org.apache.xerces.parsers.XMLParser.parse(Unknown Source)
at org.apache.xerces.parsers.AbstractSAXParser.parse(Unknown Source)
at org.xml.sax.helpers.XMLFilterImpl.parse(XMLFilterImpl.java:357)
```

part of the standard Java library and the program has been run in the Java Runtime Environment (JRE) and not the JDK.

Since EvoCrash currently requires a line number to compute the fitness values during the search, those frames have been ignored during our evaluation and do not appear in the results. Yet, as frames with missing line number appear in JCrashPack (and in other stack traces), we decided to mention this trial here as a search-based crash reproduction challenge. A possible solution, as the future work, is to relax the fitness function so that it can still approximate fitness if line numbers are missing.

Chosen example: XRENDERING-422.[34]

3.8.13 Incorrect line numbers

In 37 cases, the target frame points to the line in the source code where the target class or method is defined. This happens when the previous frame points to an anonymous class or a lambda expression. Such frames practically cannot be used for crash reproduction as the location they point to does not reveal where exactly the target exception occurs. One possible solution would be to consider the frame as having a missing line number and use the relaxed fitness function to approximate the fitness.

Chosen examples: MATH-49b, frames 1 and 4.[35]

3.8.14 Unknown

We were unable to identify why EvoCrash failed to reproduce 16 frames (out of 1,653 frames manually analyzed). In these cases, neither the logs nor the source code could help us understand how the exception was propagated.

[34] The stack trace is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/evaluation/JarFiles/resources/logs/XWIKI/XRENDERING-422/XRENDERING-422.log`

[35] See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/IrrelevantFrames.md`.

3.8.14.0.1 Summary (RQ_3) What are the open problems that need to be solved to enhance search-based crash reproduction? Based on the manual analysis of the frames that could not be reproduced at least once out of 10 rounds of executions, we identified 13 challenges for search-based crash reproduction. We confirmed challenges previously identified in other search-based software testing approaches and specified how they affect search-based crash reproduction. And discovered new challenges, more specific to search-based crash reproduction and explained how the can affect other search-based software testing approaches.

These challenges are related to the difficulty to generate test cases due to complex input data, environmental dependencies, or complex code; abstraction (static initialization, interfaces, abstract, and anonymous classes); encapsulation mechanisms (private inner classes and nested private calls in the given stack trace) of object-oriented languages; or the selection of the target frame in crash reproduction (in `try/catch` blocks, in empty enumerations, when the location in the source code is unknown, or when the frame has an incorrect line number).

3.9 Discussion

3.9.1 Empirical evaluation for crash reproduction

Conducting empirical evaluation for crash reproduction is challenging. It requires to collect various artifacts from different sources and to analyze the results to determine, in the case of a negative outcome, the cause that prevents the crash reproduction. Some are easy to fix, like missing dependencies that were added to the project linked to the stack trace, and for which we rerun the evaluation on the stack traces. The others are detailed in Section 3.8, and serve to identify future research directions.

One of the most surprising causes is due to a line mismatch in some stack traces. During the manual analysis of our results, we found out that three frames in two different stack traces, coming from Defects4J projects, target the wrong lines in the source code: the line numbers in the stack traces point to lines in the source code that cannot throw the targeted exception. Since the stack traces were collected directly from the Defects4J data (which reports failing tests and their outputs), we tried to regenerate them using the provided test suite and found a mismatch between the line numbers of the stack traces indeed. We reported those two projects to the Defects4J developers:[36] a bug in JDK7 [130] causes this mismatch. Since EvoCrash relies on

[36]See the issue at `https://github.com/rjust/defects4j/issues/142`.

line numbers to guide its search, it could not reproduce the crashes. We recompiled the source code, updated the stack trace accordingly in JCrashPack, and rerun the evaluation for those two stack traces.

Thanks to JCrashPack and ExRunner, we are now able to ease empirical evaluation for crash reproduction. ExRunner can be extended to other crash reproduction tools[37] for comparison, or assess the development of new ideas in existing tools. Our future work also includes the prioritization of crashes from JCrashPack to allow quick feedback on new ideas in a fast and automated way [43].

3.9.2 Usefulness for debugging

In our evaluation, we focused on the crash-replication capabilities of EvoCrash and identified problems affecting those capabilities. We considered the generated tests only to classify the outcomes of the EvoCrash generation process but did not assess their actual usefulness for debugging.

Chen *et al.* [81] introduced a usefulness criterion for the crash reproduction approaches. According to this criterion, a crash reproducing test is useful to the developers if it covers the buggy frame: i.e., if the target frame for which the reproduction is successful is higher than the frame that points to the buggy method.

In our previous work [204], we conducted a controlled experiment to assess the usefulness of EvoCrash for debugging and bug fixing of two crashes (one from Apache Commons Collections and one from Apache Log4j) with 35 master students. Results show that using a crash-replicating test case generated by EvoCrash may help to locate and fix the defects faster. Also, this study confirmed the usefulness criterion defined by the Chen *et al.* [81] but also found evidence that test cases categorized as not useful can still help developers to fix the bug.

Since JCrashPack also includes two open source industrial and actively maintained applications, it represents an excellent opportunity to confirm the usefulness of EvoCrash in an industrial setting. The key idea is to centralize the information in the issue tracker by providing a test case able to replicate the crash reported in an issue in the same issue (as an attachment for instance). This can be automated using, for instance, a GitHub, GitLab or JIRA plugin that executes EvoCrash when a new issue contains a stack trace. To assess the usefulness of EvoCrash in an industrial setting, we plan to setup a case study [213] with our industrial partners. Hereafter, we outline the main steps of the evaluation protocol using XWiki as subject: (i) select four crashes

[37] See how to extend ExRunner at `https://github.com/STAMP-project/ExRunner`.

to fix (two from open issues and two from closed issues) for which EvoCrash could generate a crash reproducing test for frame 3 or higher; (ii) clone the XWiki Git repository in GitHub and open four issues, corresponding to the four crash; (iii) remove the fix for the two fixed issues; (iv) for each issue, append the test case generated by EvoCrash; (v) ask (non-XWiki) developers to fix the issues; and finally, (vi) repeat the same steps without adding the test cases generated by EvoCrash (i.e., omit step iv). We would measure the time required to fix the issues (by asking participants to log that time). For the two previously fixed issues, we will compare the fixes provided by the participants with the fixes provided by XWiki developers. And for the two open issues, we will ask feedback from the XWiki developers through a pull request with the different solutions.

3.9.3 Benchmark building

JCrashPack is the first benchmark dedicated to crash reproduction. We deliberately made a biased selection when choosing Elasticsearch as the most popular, trending, and frequently-forked project from GitHub. Elasticsearch was among several other highly ranked projects, which addressed other application domains, and thus were interesting to explore. In the future, further effort should extend JCrashPack, possibly by: (i) using a *random selection* methodology for choosing projects; (ii) involving industrial projects from other application domains; and (iii) automatically collecting additional information about the crashes, the stack traces, and the frames to further understand current strengths and limitations of crash reproduction.

Building JCrashPack required substantial manual effort, not just for finding the issues, but also for collecting the right versions of the system itself and its dependencies needed to reproduce the given crash. Since we want it to be representative of current crashes, we need to automate this effort as much as possible: for instance, by mining stack traces from issue tracking systems [170].

Despite the benefits that the evaluation infrastructure could get from the inclusion of JCrashPack bugs in Defects4J, i.e., the isolation of the bugs to ease replicatbility of the evaluations [135], we designed JCrashPack as a standalone instead of extending Defects4J. The main reason is that not all bugs in Defects4J manifest as crashes (only 73 out of 395 where selected to be part of JCrashPack). We also believe that the integration of the two benchmarks is not a smooth and easy process. Defects4J requires isolation of the buggy and fixed versions of the source code, as wel as a test case able to expose the bug [135]. However, not all issues were fixed at the time we collected the crashes in JCrashPack. Also, XWiki and Elasticsearch are much larger applications

(124,000 NCSS for Elasticsearch, 177,000 NCSS for XWiki distributed in a hierarchy of several thousands of Maven projects) compared to the API libraries considered in Defects4J (63,000 NCSS for JFreeChart). Only building them with their default test suites already raised several issues. For those reasons, isolating the bug, the patch, and the non-regression test cases for such kind of large projects is not a trivial task.

3.10 Future research directions for search-based crash reproduction

From the evaluation and the challenges derived from our manual analysis, we devise the following future research directions. While the same challenge can be addressed in different ways, some requiring technical improvements of EvoCrash and other raising new research directions, we focus the discussion of this section on the latter.

3.10.1 Context matters

While search-based crash-reproduction with EvoCrash [202,204] outperformed other approaches based on (i) backward symbolic execution [81], (ii) test case mutation [**?**], and (iii) model-checking [173], our evaluation shows that the extent to which crashes are reproduced varies. These results indicate the need for taking various types of contexts and properties of software applications into account when devising an approach to a problem. Thus, we show that indeed, rather than seeking a universal approach to search-based crash reproduction, it is important to find out and address challenges specific to various types of application domains (e.g., RESTful microservices vs. enterprise wiki applications) [49].

Furthermore, search-based crash replication boils down to seeking the execution path that will reproduce a given stack trace. As with other search-based testing approaches, it faces challenges about *input data generation* during the search when the input space is large. Previous research on *mocking* and *seeding* [56, 192] address this problem by using functional mocking and extracting objects and constants from the bytecode.

We believe that *taking context into account* should go one step further for crash replication. With the development of DevOps [189] and continuous integration and delivery pipelines, there is an increasing amount of available data on the execution of the software. Those data can be used to guide the search more accurately. For instance, by seeding the search using values observed in the execution logs and setting up values for *environmental dependencies* (databases, external services, etc.).

3.10.2 Stack trace preprocessing and target frame selection

Various factors may influence the selection of a target frame in a stack trace. As observed in our evaluation, when not performed cautiously, this selection leads to unsuccessful executions of EvoCrash. For instance, frames targeting code in a *private inner class*, or *irrelevant* source code location (like, as we observed, class header or annotation) should be discarded before performing the selection.

Frames targeting code in *abstract classes* or *interfaces* (only if the target method is defined in the interface, which is possible from Java 8) may be of some use to find the cause of the crash: for instance, to identify an incorrect subclass implementation [154]. However, as abstract classes and interfaces cannot be directly instantiated, the stack trace generated by EvoCrash can never be exactly the same as the given stack trace. And, as for *input arguments* and *generic type parameters*, EvoCrash has no indication on which subclass to pick, making the search difficult. In this case, considering higher level frames (i.e., frames that are lower in the stack trace) may help to pick the right subclass.

Those reasons motivate the need to develop *stack trace analysis techniques* in order to help the *selection* of a target frame. This analysis will discard irrelevant and unknown source location frames and provide a visualization to the developer to have a clear view on what are his or her options, for instance by marking stack traces that point to interfaces and abstract classes and recommend him to pick higher level frames.

For a given stack trace, this analysis will also identify frames pointing to a *try/catch* block. Those stack traces are commonly reported by users to issue tracking systems but cannot (for now) be completely reproduced by EvoCrash. Further investigation on current error handling practices in Java code [75, 86] and how they are reported by users [157] will help us to devise efficient approaches to replicate such stack traces.

3.10.3 Guided search

Besides usage of contextual information to enhance the generation of test cases during the search process, we also consider to enhance the guidance itself. Search based testing algorithms have several parameters (365 in EvoCrash), like population size, search budget, probability of applying crossover and mutation, etc. As demonstrated by Arcuri and Fraser [51], default parameters values work well on average, but may be fare from optimal for specific frames and stack traces. A better characterization of the stack traces in JCrashPack, trying different *parameters*, as well as improving the *fitness function* itself are part of our future work. For instance the fitness func-

tion could take other elements into account (e.g., compute a similarity for exception messages). We will also consider multi-objectives search, where, for a given target frame, reproducing each lower frame becomes an objective of the search. We plan to reuse our evaluation infrastructure to compare those different approaches and investigate their different fitness landscapes to gain deeper understanding of the search process for crash reproduction. And eventually devise guidelines on EvoCrash settings to maximize crash reproduction for a given stack trace and its characteristics.

3.10.4 Improving testability

Finally, as we observed, code complexity was among the major challenges in crash reproduction with EvoCrash. To improve testability, several testability transformation techniques [**?**, 61, 118, 119, 152] have been proposed in the literature so far. Future research may investigate testability transformation techniques and their impact on search-based crash reproduction.

3.11 Threats to validity

Evaluations of crash reproduction approaches, such as the one we conducted for EvoCrash, come with threats to internal validity, external validity, and reliability. The overarching goal of JCrashPack is to reduce such threats for all evaluations of any crash reproduction tool, by offering a curated set of crashes to conduct such evaluations.

Concerning *external validity*, we carefully designed JCrashPack so that it offers a mix of small and large systems, as well as of different types of exceptions. Furthermore, it includes open source systems directly developed by industry. Nevertheless, any set is incomplete, which is why we keep JCrashPack open for extension, as discussed in Section 4.6. For example, there still remain several other domains, such as gaming or financial applications, for which there is no representative project in the benchmark.

With respect to *internal validity*, implementation faults can be a source of confounding factors. These can occur in the tools themselves, such as EvoCrash or EvoSuite, but also in the infrastructure used to actually conduct the experiment. To address the latter, JCrashPack comes with ExRunner, which automates the process of scheduling, executing, monitoring, and reporting crash reproduction attempts.

Concerning *reliability*, JCrashPack and ExRunner make it easy to repeat experiments, thus making it possible for researchers to independently replicate each others crash

reproduction findings.

Besides these threats partially mitigated by JCrashPack, our evaluation of EvoCrash comes with additional threats to (internal and external) validity. This particularly relates to the randomized nature of genetic algorithms, which we addressed by running the evaluations 10 times, and following the guidelines by Arcuri and Briand [50] for analyzing the results. Furthermore, such threats concern the risk of bias during the manual analysis, which we mitigated by using cross-checking: the result of each manual analysis has been validated by at least one other person. In case of disagreement, we asked for a third opinion. Finally, our evaluation includes only one tool: EvoCrash. Previous work showed that EvoCrash performs better than other state-of-the-art crash reproduction tools. Unfortunately, since to the best of our knowledge, no other tool was publicly available, we were not able to confirm that conclusion on the crashes in JCrashPack. We believe that JCrashPack enhances the current state-of-the-practice in crash reproduction research by offering a publicly available benchmark for which other tool providers can report their results.

3.12 Conclusion

Experimental evaluation of crash reproduction research is challenging, due to the computational resources needed by reproduction tools, the difficulty of finding suitable real life crashes, and the intricacies of executing a complex system so that the crash can be reproduced at all.

To remedy this problem, this paper sets out to create a benchmark of Java crashes, that can be reused for experimental purposes. To that end we propose JCrashPack and ExRunner, a curated benchmark of 200 real life crashes, and a tool to conduct massive experiments on these crashes. This benchmark is publicly available and can be used to compare existing and new tools against each other, as well as to analyze how proposed improvements to existing reproduction techniques actually constitute an improvement.

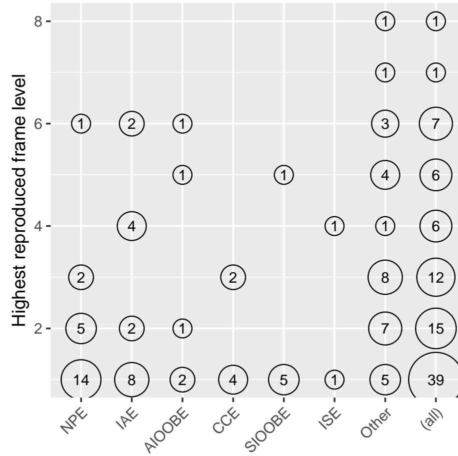
We applied the state of the art search-based Java crash reproduction tool, EvoCrash, to JCrashPack. Our findings include that the state of the art can reproduce 87 crashes out of 200 in a majority of time, that crash reproduction for industry-strength systems is substantially harder, and that `NullPointerException`s are generally easiest to reproduce. Furthermore, we identified 13 challenges that crash reproduction research needs to address to strengthen uptake in practice, as well a future research directions to address those challenges.

JCrashPack can be extended in various ways: by including more crashes from other types of applications; by automating the collection of information about eh crashes and stack traces to further understand current strengths and limitations of crash reproduction; as well as automating the collection of the crashes themselves. Furthermore, since executing crash reproduction tools on 200 crashes may be time taking, JCrashPack could be extended to offer prioritization for benchmarks, based on the known theoretical strengths and limitations if the tools. For instance, by ordering crashes based on the cyclomatic complexity of the involved frames to evaluate search-based or symbolic execution-based crash reproduction approaches.

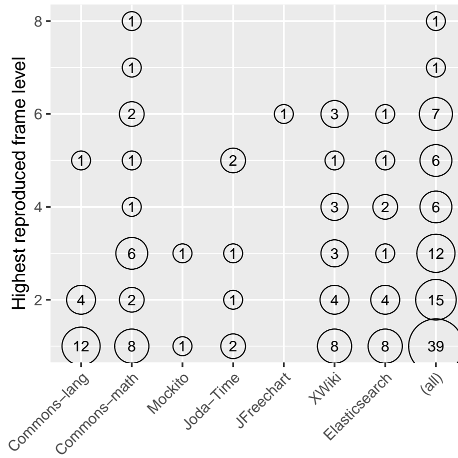
Finally, our future work for EvoCrash itself include improving input data generation by taking information from the execution context and the application (e.g., existing source code and test cases) into account. We also want to deeper our understanding of stack traces in order to be able to recommend target frames to the developers. Finally, we will improve the search process itself by refining the fitness function to improve the guidance trough the different frames of the stack trace.

Table 3.4: Number of stack traces (*st*), total number of frames (*fr*), and average number of frames ($\overline{fr}$) and standard deviation (σ) per stack trace for the different exceptions: NullPointerException (NPE), IllegalArgumentException (IAE), ArrayIndexOutOfBoundsException (AIOOBE), ClassCastException (CCE), StringIndexOutOfBoundsException (SIOOBE), IllegalStateException (ISE), and other exceptions types (Other).

Applications		**NPE**	**IAE**	**AIOOBE**	**CCE**	**SIOOBE**	**ISE**	**Other**	**Total**
Commons-lang	*st*	5.0	3.0	2.0	0.0	6.0	0.0	6.0	22.0
	fr	8.0	3.0	12.0	0.0	10.0	0.0	12.0	45.0
	$\overline{fr}$	1.6	1.0	6.0		1.7		2.0	2.0
	σ	0.9	0.0	5.7		1.0		1.5	2.1
Commons-math	*st*	3.0	3.0	4.0	2.0	1.0	0.0	14.0	27.0
	fr	8.0	7.0	9.0	11.0	1.0	0.0	70.0	106.0
	$\overline{fr}$	2.7	2.3	2.2	5.5	1.0		5.0	3.9
	σ	0.6	1.5	2.5	6.4	NA		3.0	3.0
Mockito	*st*	2.0	0.0	2.0	2.0	0.0	0.0	8.0	14.0
	fr	3.0	0.0	12.0	2.0	0.0	0.0	48.0	65.0
	$\overline{fr}$	1.5		6.0	1.0			6.0	4.6
	σ	0.7		7.1	0.0			3.8	4.1
Joda-Time	*st*	0.0	3.0	0.0	0.0	0.0	0.0	5.0	8.0
	fr	0.0	5.0	0.0	0.0	0.0	0.0	26.0	31.0
	$\overline{fr}$		1.7					5.2	3.9
	σ		0.6					1.5	2.2
JFreechart	*st*	1.0	1.0	0.0	0.0	0.0	0.0	0.0	2.0
	fr	6.0	6.0	0.0	0.0	0.0	0.0	0.0	12.0
	$\overline{fr}$	6.0	6.0						6.0
	σ	NA	NA						0.0
XWiki	*st*	20.0	4.0	0.0	6.0	1.0	0.0	20.0	51.0
	fr	535.0	39.0	0.0	131.0	8.0	0.0	687.0	1400.0
	$\overline{fr}$	26.8	9.8		21.8	8.0		34.4	27.5
	σ	33.3	3.7		22.2	NA		47.0	37.0
Elasticsearch	*st*	18.0	10.0	6.0	0.0	1.0	7.0	34.0	76.0
	fr	222.0	152.0	102.0	0.0	15.0	135.0	717.0	1343.0
	$\overline{fr}$	12.3	15.2	17.0		15.0	19.3	21.1	17.7
	σ	9.8	9.2	18.0		NA	11.9	13.4	12.5
Total	*st*	49.0	24.0	14.0	10.0	9.0	7.0	87.0	200.0
	fr	782.0	212.0	135.0	144.0	34.0	135.0	1560.0	3002.0
	$\overline{fr}$	16.0	8.8	9.6	14.4	3.8	19.3	17.9	15.0
	σ	23.9	8.5	13.3	19.3	4.8	11.9	26.3	22.3



(a) In each type of exception



(b) In each type of application

Figure 3.4: Highest reproduced frame levels

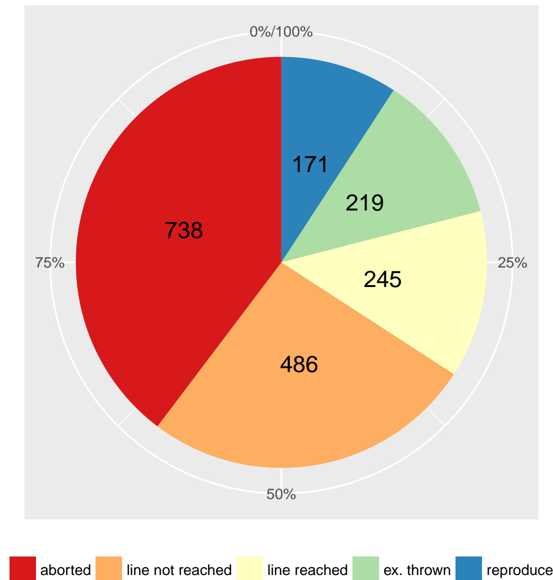


Figure 3.5: An overview of the reproduction outcome

Table 3.5: Statistics for the average number of fitness evaluations for the *reproduced* frames (**fr**) belonging to different stack traces (**st**), grouped by **applications**, out of 10 rounds of execution. The confidence Interval (**CI**) is calculated for the median bootstrapping with *100,000* runs, at a 95% confidence level.

Applications	st	fr	Min	Lower Quart.	Median CI	Med.	Upper Quart.	Max
Com.-lang	19	213	1	2.0	[5.0 ,22.0]	15.0	237.0	52,240
Com.-math	24	471	1	13.0	[124.0 ,211.0]	178.0	1,046.5	58,731
Mockito	2	40	1	1.0	[1.0 ,1.0]	1.0	5.2	138
Joda-Time	6	138	1	15.5	[79.1 ,369.0]	253.5	1,290.2	40,189
JFreechart	1	41	1	10.0	[-292.0 ,350.0]	221.0	1,188.0	20,970
XWiki	25	531	1	2.5	[14.0 ,30.0]	23.0	209.0	34,089
Elasticsearch	19	287	1	4.0	[5.0 ,32.0]	23.0	125.0	17,461
Total	96	1721	1	4.0	[34.0 ,59.0]	48.0	534.0	58,731

Figure 3.6: Detailed reproduction outcome for the different frames.

Table 3.6: Statistics for the average number of fitness evaluations for the *reproduced* frames (**fr**) belonging to different stack traces (**st**), grouped by **exceptions**, out of 10 rounds of execution. Confidence Interval (**CI**) is calculated for median with bootstrapping with *100,000* runs, at 95% confidence level.

Applications	st	fr	Min	Lower Quart.	Median CI	Med.	Upper Quart.	Max
NPE	26	330	1	6.0	[9.0 ,63.0]	44.5	220.0	34,089
IAE	16	399	1	2.0	[7.0 ,12.0]	10.0	49.0	38,907
AIOOBE	5	58	1	15.5	[252.0 ,1,104.5]	675.0	1,671.2	53,644
CCE	6	103	1	6.5	[74.0 ,210.0]	120.0	560.0	10,197
SIOOBE	8	95	1	12.5	[122.0 ,945.0]	505.0	2,326.0	52,240
ISE	2	42	1	1.0	[1.0 ,3.0]	2.0	105.8	1,138
Other	33	694	1	7.0	[99.0 ,139.0]	125.5	825.0	58,731
Total	96	1721	1	4.0	[34.0 ,59.0]	48.0	534.0	58,731

Table 3.7: Challenges with the number and percentage of frames identified for this challenge.

Category	**Frames**	**Frequency**
Input Data Generation	825	49.91%
Abstract Class	242	14.64%
Anonymous Class	142	8.59%
Static Initialization	141	8.53%
Complex Code	118	7.14%
Private Inner Class	56	3.39%
Environmental Dependencies	52	3.15%
Irrelevant Frame	37	2.24%
Unknown Sources	16	0.97%
Nested calls	10	0.60%
try/catch	7	0.42%
Interface	6	0.36%
Empty Enum Type	1	0.06%
Total	1653	100%

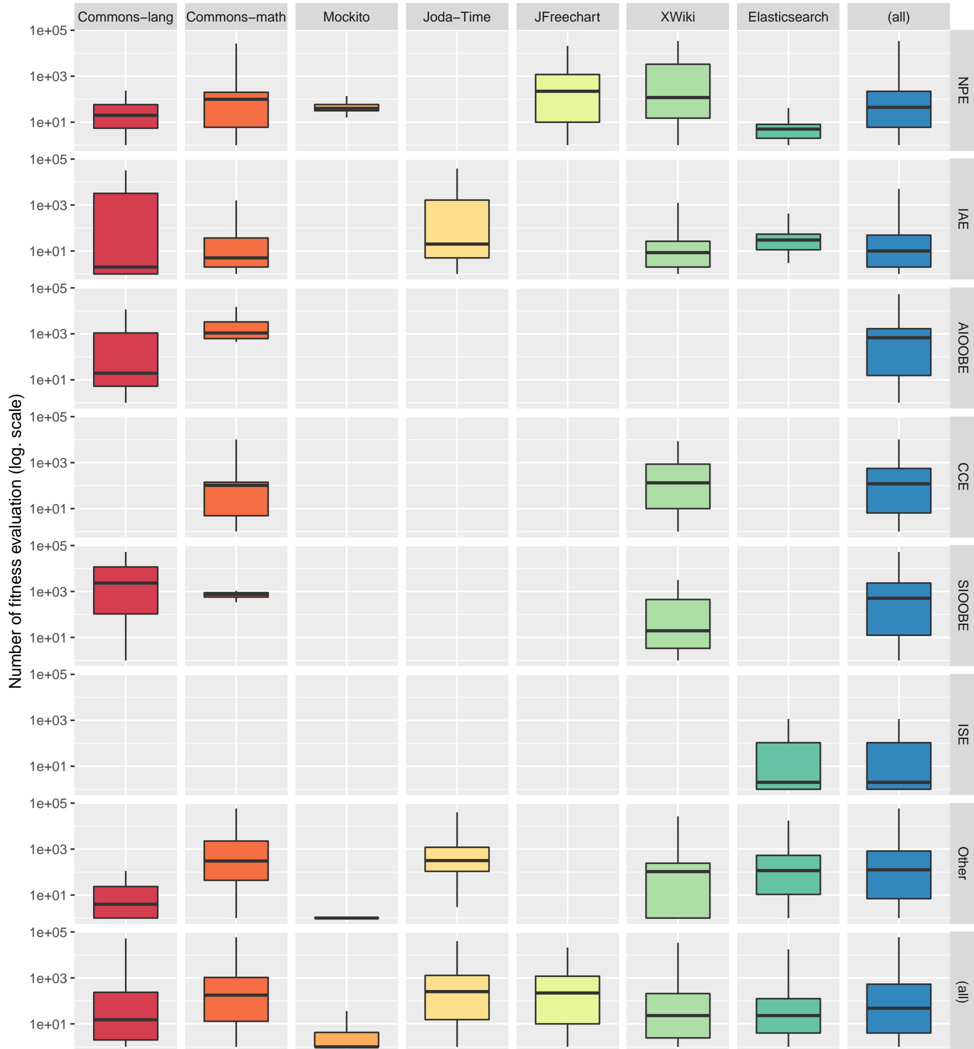


Figure 3.7: Average number of fitness evaluations for the *reproduced* frames for each applications and exception type.

4 Fitness Function Evaluation

EvoCrash is a recent search-based approach to generate a test case that reproduces reported crashes. The search is guided by a fitness function that uses a weighted sum scalarization to combine three different heuristics: (i) code coverage, (ii) crash coverage and (iii) stack trace similarity. In this study, we propose and investigate two alternatives to the weighted sum scalarization: (i) the simple sum scalarization and (ii) the multi-objectivization, which decomposes the fitness function into several optimization objectives as an attempt to increase test case diversity. We implemented the three alternative optimizations as an extension of EvoSuite, a popular search-based unit test generator, and applied them on 33 real-world crashes. Our results indicate that for complex crashes the weighted sum reduces the test case generation time, compared to the simple sum, while for simpler crashes the effect is the opposite. Similarly, for complex crashes, multi-objectivization reduces test generation time compared to optimizing with the weighted sum; we also observe one crash that can be replicated only by multi-objectivization. Through our manual analysis, we found out that when optimizing the original weighted function gets trapped in local optima, optimization for decomposed objectives improves the search for crash reproduction. Generally, while multi-objectivization is under-explored, our results are promising and encourage further investigations of the approach.

4.1 Introduction

Crash reproduction is an important step in debugging field crashes. Therefore, various automated approaches to crash reproduction [65, 81, 173, 194, 202, 215] have been proposed in the literature. Among these, EvoCrash [202] is a search-based approach, which applies a Guided Genetic Algorithm (GGA) to generate a crash-reproducing test. To optimize test generation for crash reproduction, the GGA uses a weighted-sum scalarized function, which is a sum of three heuristics, namely: (i) line coverage, (ii) exception coverage, and (iii) stack trace similarity rate. The function resulting from the sum scalarization is further subject to the constraint that the target exception has to be thrown at the code line reported in the crash stack trace. Depending on how close a generated test case may come to trigger a reported crash, its fitness value may be between 0.0 (i.e., each of the three heuristics evaluates to 0.0), and 6.0 (i.e., none of the heuristics is satisfied by the generated test). Soltani et. al [202] evaluated EvoCrash on 50 real-world crashes and showed that the search-based approach improved over other non-search-based approaches proposed in the related literature [81, 173, 215].

As any search-based technique, the success of EvoCrash depends on its capability of maintaining a good balance between *exploitation* and *exploration* [88]. The former refers to the ability to visit regions of the search space within the neighborhood of the current solutions (i.e., refining previously generated tests); the latter refers to the ability to generate completely different new test cases. In crash reproduction, the exploitation is guaranteed by the guided genetic operators that focus the search on methods appearing in the crash stack trace [202]. However, such a depth and focused search may lead to a low exploration power. Poor exploration results in low diversity between the generated test cases and, consequently, the search process easily gets trapped in local optima [88].

In this paper, we investigate two strategies to increase the diversity of generated test cases for crash reproduction. While EvoCrash uses one single-objective fitness function to guide the search, prior studies in evolutionary computation showed that relaxing the constraints [87] or multi-objectivizing the fitness function [140] help promoting diversity. Multi-objectivization is the process of (temporarily) decomposing a single-objective fitness function into multiple sub-objectives to optimize simultaneously with multi-objective evolutionary algorithms. At the end of the search, the global optimal solution of the single-objective problem is one of the points of the Pareto front generated by the multi-objective algorithms. The decomposed objectives should be as independent of each other as possible to avoid getting trapped in local optima [140].

Therefore, we study whether transforming the original weighted scalarized function in EvoCrash into (i) a simple scalarized function via constraint relaxation, and (ii) multiple decomposed objectives, impacts the crash reproduction rate, and test generation time. EvoCrash [202] relies on EvoSuite [103] for test generation, and as such, we implemented the original weighted function as an extension of EvoSuite. Similarly, we implemented the alternative optimization functions by extending EvoSuite. We evaluated the alternatives on 33 real-world crashes from four open source projects. Our results show that indeed, when crashes are complex and require several generations of test cases, using multi-objectivization reduces the test generation time compared to the weighted scalarized function, and in turn, the weighted scalarized function reduces test generation time compared to the simple scalarized function. Furthermore, we observe that one crash can be fully replicated only by multi-objectivized search and not by the two single-objective strategies. Generally, our results show that problems that are single-objective by nature can benefit from multi-objectivization. We believe that our findings will foster the usage of multi-objectivization in search-based software engineering.

The remainder of the chapter is structured as follows: Section 4.2 provides background and related work. Section 4.3 describes single and multi-objectivization for crash reproduction. Sections 4.4 and 4.5 present the evaluation and results, respectively. Discussion follows in Section 4.6. Section 4.7 concludes.

4.2 Background and Related Work

Crash reproduction tools aim at generating a test case able to reproduce a given crash based on the information gathered during the crash itself. This *crash reproduction test case* can help developers to identify the fault causing the crash [81]. For Java programs, the available information usually consists of a stack trace, i.e., lists of classes, methods and code lines involved in the crash. For instance, the following stack trace has been generated by the test cases of `LANG v9b` from the Defects4J [135] dataset:

```
0 java.lang.ArrayIndexOutOfBoundsException:
1   at org.apache.commons.lang3.time.FastDateParser.toArray(FastDateParser.java:413)
2   at org.apache.commons.lang3.time.FastDateParser.getDisplayNames(FastDateParser
3     .java:381)
4   ...
```

It has a thrown exception (`ArrayIndexOutOfBoundsException`) and different frames (lines 1 to 3), each one pointing to a method call in the source code.

4.2.1 Related Work

Over the years, various Java crash replication approaches that use stack traces as input have been developed. RECORE [194] is a search-based approach that in addition to crash stack traces, uses core dumps as input data for automated test generation. MUCRASH [215] applies mutation operators on existing test cases, for classes that are present in a reported stack trace, to trigger the reported crash. While BUGREDUX [134] is based on forward symbolic execution, STAR [81] is a more recent approach that applies optimized backward symbolic execution on the method calls recorded in a stack trace in order to compute the input parameters that trigger the target crash. JCHARMING [173] is also based on using crash stack traces as the only source of information about a reported crash. JCHARMING [173] applies directed model checking to identify the pre-conditions and input parameters that cause the target crash. Finally, CONCRASH [65] is a recent approach that focuses on reproducing concurrency crashes, in particular. CONCRASH applies pruning strategies to iteratively look for test code that triggers the target crash in a thread interleaving.

More recently, Soltani et al. have proposed EVOCRASH [202], an evolutionary search-based tool for crash replication built on top of EVOSUITE [105]. EvoCrash uses a novel Guided Genetic Algorithm (GGA), which focuses the search on the method calls that appear in the crash stack trace rather than maximizing coverage as in classical coverage-oriented GAs. Their empirical evaluation demonstrated that EvoCrash outperforms other existing crash reproduction approaches.

4.2.2 EvoCrash

To design EvoCrash, Soltani et al. [202] defined a fitness function (*weighted sum fitness function*) and a search algorithm (*guided genetic algorithm*) dedicated to crash reproduction. The fitness function is used to characterize the “quality” of test case generated during each iteration of the guided GA.

4.2.2.1 Weighted Sum (WS) Fitness Function

The three components of the WS fitness function are: (i) the coverage of the code line (*target statement*) where the exception is thrown, (ii) the target exception has to be thrown, and (iii) the similarity between the generated stack trace (if any) and the

original one. Formally, the fitness function for a given test t is defined as [202]:

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_{except}) + max(d_{trace}) & \textit{if the line is not reached} \\ 3 \times min(d_s) + 2 \times d_{except}(t) + max(d_{trace}) & \textit{if the line is reached} \\ 3 \times min(d_s) + 2 \times min(d_{except}) + d_{trace}(t) & \textit{if the exception is thrown} \end{cases} \tag{4.1}$$

where $d_s(t) \in [0, 1]$ denotes how far t is from executing the target statement using two well-known heuristics, *approach level* and *branch distance* [201]. The approach level measures the minimum number of control dependencies between the path of the code executed by t and the target statement s. The branch distance scores how close t is to satisfying the branch condition for the branch on which the target statement is directly control dependent [160]. In Equation 4.1, $d_{except}(t) \in \{0, 1\}$ is a binary value indicating whether the target exception is thrown (0) or not (1); $d_{trace}(t)$ measures the similarity of the generated stack trace with the expected one based on methods, classes, and line numbers appearing in the stack traces; $max(d_{except})$ and $max(d_{trace})$ denote the maximum possible value for d_{except} and d_{trace}, respectively. Therefore, the last two addends of the fitness function (i.e., d_{except} and d_{trace}) are computed upon the satisfaction of two *constraints*. This is because the target exception has to be thrown in the target line s (first constraint) and the stack trace similarity should be computed only if the target exception is actually thrown (second constraint).

4.2.2.2 Guided Genetic Algorithm (GGA)

EvoCrash (as EvoSuite) generates test cases at the unit level, meaning that test cases are generated by instrumenting and targeting one particular class (the *target class*). Contrary to classical unit test generation, EvoCrash does not seek to maximize coverage by invoking all the methods of the target class, but privileges those involved in the target failure. This is why the GGA algorithm relies on the stack trace to guide the search and reduces the search space at different steps. (i) A *target frame* is selected by the user amongst the different frames of the input stack trace. Usually, the target frame is the last one in the crash trace as it corresponds to the root method call where the exception was thrown. The class appearing in this target frame corresponds to the target class for which a test case will be generated. (ii) The *initial population* of test cases is generated in such a way that the method m of the target frame (the *target method*) is called at least once in each test case [202]: either directly if m is public or protected, or indirectly by calling another method that invokes the target method if m is private. (iii) During the search, dedicated *guided crossover* and *guided mutation* operators [202] ensure that newly generated test cases contain at least one call to the

target method. (iv) The search is guided by the WS *fitness function*. (v) Finally, the algorithm stops if the time budget is consumed or when a zero-fitness value is achieved. In this last case, the test case is minimized by a *post-processing* that removes randomly inserted method calls that do not contribute to reproducing the crash.

4.3 Single-Objective and Multi-Objectivization for Crash Reproduction

A key limitation of evolutionary algorithms (and metaheuristics in general) is that they may become trapped in local optima due to *diversity loss* [88], a phenomenon in which no modification (with crossover and mutation) of the current best solutions will lead to discovering a better one. This phenomenon is quite common in white-box unit-level test case/suite generation, as shown by previous studies in search-based software testing [42, 99, 121, 138]. Many strategies have been investigated by the evolutionary computation community to alleviate the problem of diversity loss, including (i) combining different types of evolutionary algorithms [88, 121], (ii) defining new genetic operators to better promote diversity [88, 93, 121], (iii) altering the fitness function [88, 113, 140], and (iv) relaxing the constraints of the problem [87].

In the context of crash replication, most attention has been devoted to improving the genetic operators [201, 202] to better focus the search on method calls related to the target crash. However, to the best of our knowledge, no previous study investigated alternative formulations to the fitness function in Equation 4.1 and how they are related to diversity and convergence to local optima. The original equation by Soltani et al. [202] (i.e., Equation 4.1) combines three different factors into one single scalar value based on some constraints. Given this type of equation, there are two possible alternatives to investigate: (i) relaxing the constraints and (ii) split the fitness function into three search objectives to optimize simultaneously. The next subsections describe these two alternative formulations of the crash replication problem and how they are related to test case diversity.

4.3.1 Constraints Relaxation

As explained in Section 4.2, the crash replication problem has been implicitly formulated in previous studies as a constraint problem. The constraints are handled using *penalties* [202], i.e., the fitness score of a test case is penalized by adding (or subtracting in case of a maximization problem) a certain scalar value proportional to the

number of constraints being violated. For example, in Equation 4.1 all test cases that do not cover the target code line are penalized by the two addends $2 \times max(d_{except})$ and $max(d_{trace})$ as there are two violated constraints (i.e., the line to cover and the exception to throw in that line). Instead, tests that cover the target line but that do not trigger the target exception are penalized by the factor $max(d_{trace})$ (only one constraint is violated in this case).

While adding penalties is a well-known strategy to handle constraints in evolutionary algorithms [87], it may lead to diversity loss because any test not satisfying the constraints have very low probability to survive across the generations. For example, let us assume for example that we have two test cases t_1 and t_2 for the example crash reported in Section 4.2. Now, let us assume that both test cases have a distance $d_s = 1.0$ (i.e., none of the two could cover the target line), but the former test could generate an exception while the latter does not. Using Equation 4.1, the fitness value for both t_1 and t_2 is $f(t_1) = f(t_2) = 3 \times d_s + 3.0 = 6.0$. However, t_2 should be promoted if it can generate the same target exception of the target crash (although on a different line) and the generated trace is somehow similar to the original one (e.g., some methods are shared).

Therefore, a first alternative to the fitness function in Equation 4.1 consists of relaxing the constraints, i.e., removing the penalties. This can be easily implemented with a *Simple Sum Scalarization* (SSS):

$$f(t) = d_s(t) + d_{except}(t) + d_{trace}(t) \tag{4.2}$$

where $d_s(t)$, $d_{except}(t) \in \{0, 1\}$, and $d_{trace}(t)$ are the same as in Equation 4.1. This relaxed variant —hereafter referred as *simple sum scalarization*— helps increase test case diversity because test cases that lead to better $d_{except}(t)$ or $d_{trace}(t)$ may survive across the GGA generation independently from the value of $d_s(t)$, which was not the case for the weighted sum, thanks to the constraints from Equation 4.1. On the other hand, this reformulation may increase the number of local optima; therefore, an empirical evaluation of weighted and simple sum variants to the fitness function is needed.

4.3.2 Multi-objectivization

Knowles et al. [140] suggested to replace the original single-objective fitness function of a problem with a set of new objectives in an attempt to promote diversity. This process, called *multi-objectivization* (MO), can be performed in two ways [127, 140]: (i) by decomposing the single-objective function into multiple sub-objectives, or (ii)

by adding new objectives in addition to the original function. The multi-objectivized problem can then be solved using a multi-objective evolutionary algorithm, such as NSGA-II [93]. By definition, multi-objectivization preserves the global optimal solution of the single-objective problem that, after problem transformation, becomes a Pareto efficient solution, i.e., one point of the Pareto front generated by multi-objective algorithms.

In our context, applying multi-objectivization is straightforward as the fitness function in Equation 4.1 is defined as the weighted sum of three components. Therefore, our multi-objectivized version of the crash replication problem consists of optimizing the following three objectives:

$$\begin{cases} f_1(t) = d_s(t) \\ f_2(t) = d_{except}(t) \\ f_3(t) = d_{trace}(t) \end{cases} \tag{4.3}$$

Test cases in this three-objectivized formulation are therefore compared (and selected) according to the concept of *dominance* and *Pareto optimality*. A test case t_1 is said to *dominate* another test t_2 ($t_1 \prec_p t_2$ in math notation), iff $f_i(t_1) \leq f_i(t_2)$ for all $i \in \{1, 2, 3\}$ and $f_j(t_1) < f_j(t_2)$ for at least one objective f_j. A test case t is said *Pareto optimal* if there does not exist any another test case t_3 such that $t_3 \prec_p t_1$. For instance, for the test cases (i.e., solutions) generated by a multi-objectivized (*Multi-obj.*) search presented in Figure 4.1, A, B, and D dominate C, E, and F.

In our problem, there can be multiple non-dominated solutions within the population generated by GGA at a given generation. These non-dominated solutions represent the best trade-offs among the search objectives that have been discovered/generated during the search so far. Diversity is therefore promoted by considering all non-dominated test cases (trade-offs) as equally good according to the *dominance* relation and that are assigned the same probability to survive in the next generations.

It is worth noting that a test case t that replicates the target crash will achieve the score $f_1(t) = f_2(t) = f_3(t) = 0$, which is the optimal value for all objectives. In terms of optimality, t is the global optimum for the original single-objective problem but it is also the single Pareto optimal solution because it dominates all other test cases in the search space. This is exactly the main difference between classical multi-objective search and multi-objectivization: in multi-objective search we are interested in generating a well-distributed set of Pareto optimal solutions (or optimal trade-offs); in multi-objectivization, some trade-offs are generated during the search (and preserved to help diversity), but there is only one optimal test case, i.e., the one reproducing the target crash.[1]

[1]Note that there might exist multiple tests that can replicate the target crash; however, these tests are

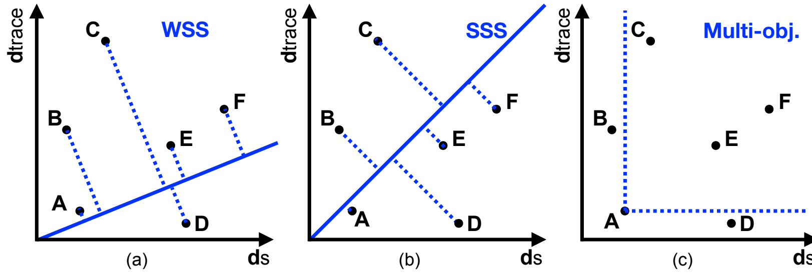


Figure 4.1: A Graphical Interpretation of Different Fitness Functions

Non-dominated Sorting Genetic Algorithm II. To solve our multi-objectivized problem, we use NSGA-II [93], which is a well-known multi-objective genetic algorithm (GA) that provides well-distributed Pareto fronts and good performance when dealing with up to three objectives [93]. As any genetic algorithm, NSGA-II evolves an initial population of test cases using crossover and mutation; however, differently from other GAs, the selection is performed using tournament selection and based on the *dominance* relation and the *crowding distance*. The former plays a role during the *non-dominated sorting* procedure, where solutions are ranked in non-dominance fronts according to their dominance relation; non-dominated solutions have the highest probability to survive and to be selected for reproduction. The crowding distance is further used to promote the more diverse test cases within the same non-dominance front.

In this paper, we implemented a *guided* variant of NSGA-II, where its genetic operators are replaced with the *guided crossover* and *guided mutation* implemented in GGA. We used these operators (i) to focus the search on the method call appearing in the target trace and (ii) to guarantee a fair comparison with GGA by adopting the same operators.

4.3.3 Graphical Interpretation

Figure 4.1 shows commonalities and differences among the tree alternative formulations of the crash reproduction problem (see sections 4.3.1 and 4.3.2). For simplicity, let us focus on only two objectives (d_s and d_{trace}) and let us assume that we have a set of generated tests which are shown as points in the bi-dimensional space delimited by the two objectives. As shown in Figure 4.1(c), points (test cases) in multi-objectivization are compared in terms of non-dominance. In the example, the tests

coincident points as they will all have a zero-value for all objectives.

A, B, and D are non-dominated tests and all of them are assigned to the first non-dominance front in NSGA-II, i.e., they have the same probability of being selected. On the other hand, sum scalarization (either simple or weighted) projects all point to one single vector, i.e., the blue lines in Figures 4.1(a) and 4.1(b). With weighted sum scalarization (WSS), the vector of the aggregated fitness function is inclined to the d_s axis due to the higher weight of the line coverage penalty. In contrast, the vector obtained with simple sum scalarization (SSS) is the bisector of the first quadrant, i.e., both objectives share the same weights. While in both Figure 4.1(a) and 4.1(b), the best solution (point A) is the one closer to the origin of the axes, the order of the solutions (and their selection probability) can vary. For instance, we can see in the Figure that case C is a better choice than case D in the weighted sum because it has a lower value for d_s. But, case D is better than C in the simple sum. These differences in the selection procedure may lead the search toward exploring/exploiting different regions of the search space.

4.4 Empirical Evaluation

We conducted an empirical evaluation to assess the impact of the single objective or multi objectivization fitness functions, answering the following research questions:

RQ$_1$ *How does crash reproduction with simple sum scalarization compare to crash reproduction using weighted sum scalarization?*

RQ$_2$ *How does crash reproduction with a multi-objectivized optimization function compare to crash reproduction using weighted sum scalarization?*

Comparisons for **RQ**$_1$ and **RQ**$_2$ are done by considering the number of crashes reproduced (*crash coverage rate*) and the time taken by EvoCrash to generate a crash reproducing test case (*test generation time*).

4.4.1 Setup

To perform our evaluation, we randomly selected 33 crashes from five open source projects: 18 crashes from four projects contained in Defects4J [135], which is a well-known collection of bugs from popular libraries; and 12 crashes from XWiki,[2] a web application project developed by our industrial partner.

[2]http://www.xwiki.org/

Table 4.1: Crashes used in the study.

Exception Type	Defects4J	XWiki
`NullPointerException` (NPE)	9	9
`ArrayIndexOutOfBoundsExceptions` (AIOOBE)	7	0
`ClassCastException` (CCE)	2	3

We execute the EvoSuite extensions, with the three approaches (weighted sum, simple sum, and multi-objectivization), on 23 virtual machines. Each machine has 8 CPU-cores, 32 GB of memory, and a 1TB shared hard drive. All of them run CentOs Linux release 7.4.1708 as operating system, with OpenJDK version 1.8.0-151.

For each crash c, we run each approach in order to generate a test case that reproduces c and targeting each frame one by one, starting from the highest one (the last one in the stack frame). As soon as one of the approaches is able to generate a test case for the given frame (k), we stop the execution and do not try to generate test cases for the lower frames ($< k$). To address the random nature of the evaluated search approaches, we execute each approach 15 times on each frame for a total number of 12,022 executions independent runs.

Parameter settings. We use the default parameter configurations from EvoSuite with functional mocking to minimize the risk of environmental interactions and increase the coverage [56]. We set the search budget to 10 minutes, which is double of the maximal amount reported by Soltani et al. [202].

4.4.2 Analysis

Since the crash coverage data is a binary distribution (i.e., a crash is reproduced or not), we use the Odds Ratio (OR) to measure the impact of the single or multi-objectivization on the *crash coverage* rate. A value of OR $>$ 1 for comparing a pair of factors (A, B) indicates that the coverage rate increases when factor A is applied, while a value of OR $<$ 1 indicates the opposite. A value of OR $=$ 1 indicates that there is no difference between A and B. In addition, we use Fisher's exact test, with α=0.05 for Type I errors to assess the significance of the results. A p-value $<$ 0.05 indicates the observed impact on the coverage rate is statistically significant, while a value of p-value $>$ 0.05 indicates the opposite.

Furthermore, we use the Vargha-Delaney $\hat{A}_{12}$ statistic [207] to assess the effect size of the differences between the two sum scalarization approaches or between weighted sum and multi-objectivization for *test generation time*. A value of $\hat{A}_{12} < 0.5$ for a pair

of factors (A, B) indicates that A reduces the test generation time, while a value of $\hat{A}_{12} > 0.5$ indicates that B reduces the generation time. If $\hat{A}_{12} = 0.5$, there is no difference between A and B on generation time. To check whether the observed impacts are statistically significant, we used the non-parametric Wilcoxon Rank Sum test, with α=0.05 for Type I error. *P*-values smaller than 0.05 indicate that the observed difference in the test generation time is statistically significant.

4.5 Results

In this section, we present the results of the experiments. Thereby, we answer the two research questions on comparing simple and weighted sum aggregation functions as well as weighted sum and multi-objectivization for crash reproduction.

Results (RQ1). Table 4.2 (please see the end of the chapter) presents the crash reproduction results for the 33 crashes used in the experiment. As the table shows, 21 cases were reproduced using the original weighted sum scalarized function, while 20 cases were reproduced using simple sum scalarization. Thus, `MATH-32b` is only reproduced by the weighted sum approach. Both optimization approaches reproduced the crashes at the same frame level.

As Table 4.3 (please see the end of the chapter) shows, we do not observe any statistically significant impact on the crash reproduction rate, comparing weighted and simple sum scalarization. However, for one case, `XWIKI-13031`, the odds ratio measure is 6.5, which indicates that the rate of crash reproduction using the weighted scalarized function is 6.5 times larger than the reproduction rate of using the simple scalarized function. In this case, the p value is 0.1, therefore we cannot draw a statistically significant conclusion.

For four cases, we see a significant impact on the test generation time. Based on our manual analysis, we observe that when a crash (`XWIKI-13031`) is complex, i.e., it takes several generations to produce a crash reproducing test case, weighted sum reduces execution time. However, when a crash, e.g., `XWIKI-13377`, is easy to reproduce, then weighted sum takes longer to find a crash reproducing test.

Results (RQ2). Table 4.2 shows that 22 cases were reproduced using decomposed crash optimization objectives, while 21 cases were reproduced by the original weighted sum function. `XWIKI-14475` is reproduced by the multi-objectivized approach only.

As Table 4.3 shows, in most cases, we do not observe any impact on the rate of crash coverage. However, for `MATH-81b` and `LANG-57b`, the odds ratio measures are

4.8 and 1.7 respectively, which indicates that the rate of crash reproduction using multi-objectivized optimization is 4.8 times and 1.7 times higher than the rate of reproduction using the weighted sum function. For these cases, the *p*-values are 0.3 and 0.6 respectively, therefore, we cannot draw a statistically significant conclusion yet.

Moreover, as Table 4.3 shows, for six cases, namely: `MATH-100b, MATH-32b, MATH-4b, MATH-98b, XWIKI-13031`, and `XWIKI-14319`, we observe that using multi-objectivization reduces the time for test generation (as $\hat{A}_{12}$ measures are lower than 0.5). For all these cases, the *p* values are lower than 0.05, which indicates the observed impacts are statistically significant. On the other hand, for four other cases, namely: `LANG-33b, LANG-39b, LANG-47b`, and `MATH-70b`, we observe an opposite trend, i.e., the weighted sum achieves a lower test generation time (as the $\hat{A}_{12}$ measures are larger than 0.5). Based on our manual analysis, as also indicated by the average execution time values reported in Table 4.2, when a crash is complex and the search requires several generations (e.g., `XWIKI-13031`), multi-objectivization reduces the execution time. On the other hand, when a crash is easy to be reproduced and a few generations of test cases quickly converge to a global optimum, then using the weighted sum approach is more efficient.

4.6 Discussion

As Table 4.3 shows, for only one case, `XWIKI-13031`, the weighted sum is more efficient than the simple sum, while for two other cases, `XWIKI-13377` and `CHART-4b`, the simple sum is more efficient. From our manual analysis of these cases, we see that when the target line is covered in a few seconds (when initializing the first population), the simple sum is more efficient than the weighted sum. However, when more search iterations (generations) are needed to find a test that reaches the target line, like for `XWIKI-13031`, the weighted sum is much faster. As indicated in Section 4.3, while using weights in single-objective optimization may reduce the likelihood of getting stuck in local optima, it may accept solutions that trigger the target exception but not at the target code line. Therefore, a possible explanation for these cases is that while maintaining diversity improves efficiency to a small degree, relaxing the constraints may penalize the exploitation. In practice, since it is not possible to know *a priori* when getting stuck in local optima occurs, using weighted sum (that provides more guidance, thanks to the constraints it takes into account) seems a more reliable approach, which might be few seconds less efficient compared to simple sum (in some cases).

As Knowles et. al [140] discussed, when applying multi-objectivization, for a successful search, it is important to derive independent objectives. In our multi-objectivization approach, as presented in Section 4.3, we decompose the three heuristics in the original scalarized function into three optimization objectives. However, these objectives are not entirely independent of each other; line coverage is interrelated to the stack trace similarity. Thus, if the target line is not covered, the stack trace similarity will never converge to 0.0. This can be one possible explanation for why when the target frame is one, single-objective optimization performed better for most cases in our experiments. The fewer frames to reproduce, the stronger the interrelation between the two objectives is.

Furthermore, we observe that when a crash is complex and requires several generations to be reproduced, the multi-objectivized approach performs more efficiently than single-objective optimization. On the other hand, when crashes can be reproduced in few generations (i.e., the target line is covered by the initial population of GAs and evolution is mostly needed for triggering the same crash), then the single-objective approach is more efficient. This is due to the cost of the fast non-domination sorting algorithm in NSGA-II [93], whose computational complexity is $\mathcal{O}(MN^2)$, where M is the number of objectives and N is the population size. Instead, the computational complexity of the selection in a single-objective GA is $\mathcal{O}(M)$, where N is the population size. Thus, sorting/selecting individuals is computationally more expensive in NSGA-II and it is worthwhile only when converging to 0.0 requires effective exploration through the enhanced diversity in NSGA-II.

Insights. From our results and discussion, we formulate the following insights: (i) **prefer multi-objectivization**, as it substantially reduces the execution time for complex crashes (up to three minutes) and the time loss for simple crashes is small (few seconds on average); furthermore, it allows to reproduce one additional crash that weighted sum could not reproduce; (ii) Alternatively, use a **hybrid search** that switches from weighted sum to multi-objectivized search when the execution time is above a certain threshold (20 seconds in our case) or if the target code line is not covered within the first few generations; and finally, (iii) Avoid **simple sum scalarization** as it may get stuck into local optima (multi-objectivization).

Threats to validity. We randomly selected 33 crashes from five different open source projects for our evaluation. Those crashes come from Defects4J, a collection of defects from popular libraries, and from the issue tracker of our industrial partner, ensuring diversity in the considered projects. In addition, the selected crashes contain three types of commonly occurring exceptions. While we did not analyze the exception types, they may be a factor that impacts the test generation time and crash reproduc-

tion rate. Finally, our extension to EvoSuite may contain unknown defects. To mitigate this risk, in addition to testing the extensions, the first three authors reviewed the artifacts independently.

4.7 Conclusion

Crash reproduction is an important step in the process of debugging field crashes that are reported by end users. Several automated approaches to crash reproduction have been proposed in the literature to help developers debug field crashes. EvoCrash is a recent approach which applies a Guided Genetic Algorithm (GGA) to generate a crash reproducing test case. GGA uses a weighted scalarized function to optimize test generation for crash reproduction. In this study, we apply the GGA approach as an extension of EvoSuite and show that using a weighted sum scalarization fitness function improves test generation compared to a simple sum scalarization fitness function when reproducing complex crashes. Moreover, we also investigate the impact of decomposing the scalarized function into multiple optimization functions. Similarly, compared to using the weighted scalarized function, we observe that applying multi-objectivization improves the test generation time when reproducing complex crashes requiring several generations of test case evolution.

In general, we believe that multi-objectivization is under-explored to tackle (by-nature-) single-objective problems in search-based software testing. Our results on multi-objectivization by decomposition of the fitness function for crash reproduction are promising. This calls for the application of this technique to other (by-nature-) single-objective search-based problems.

Table 4.2: Experiment results for Multi-objectivized (Multi-obj.), Weighted (WSS) and Simple Sum (SSS) Scalarization. "-" indicates that the optimization approach did not reproduce the crash. Bold cases represent the crashes only reproduced by some of the approaches, not all. **Rep., T.,** and **SD** indicate reproduction rate, average execution time, and standard deviation, respectively.

			Multi-obj.			WSS			SSS		
Crash ID	**Exception**	**Frame**	Rep.	$\overline{T}$	SD	Rep.	$\overline{T}$	SD	Rep.	$\overline{T}$	SD
CHART-4b	NPE	6	15	16.5	1.4	15	16.6	1.4	15	14.8	1.3
LANG-12b	AIOOBE	2	15	2.5	0.3	15	2.5	0.5	15	2.4	0.5
LANG-33b	NPE	1	15	1.7	0.0	15	1.0	0.2	15	1.0	0.0
LANG-39b	NPE	2	15	2.7	1.0	15	1.1	0.5	15	1.6	1.2
LANG-47b	NPE	1	15	3.4	1.3	15	2.1	1.1	15	1.0	0.7
LANG-57b	NPE	1	11	1.1	0.0	9	185.0	288.0	12	86.1	218.1
LANG-9b	AIOOBE	-	-			-			-		
MATH-100b	AIOOBE	1	15	8.4	13.4	15	7.2	1.7	15	8.2	7.3
MATH-32b	**CCE**	**1**	**15**	**3.9**	**0.9**	**15**	**5.3**	**2.5**	**-**		
MATH-4b	NPE	3	15	27.3	49.2	14	21.7	16.1	14	62.0	150.0
MATH-70b	NPE	3	15	1.7	0.2	15	1.1	0.3	15	1.0	0.0
MATH-79b	NPE	1	15	1.7	0.1	15	1.0	0.2	15	1.0	0.0
MATH-81b	AIOOBE	6	9	82.0	63.0	11	180.7	230.5	15	115.0	114.0
MATH-98b	AIOOBE	1	15	7.7	5.3	14	9.5	5.7	15	9.9	9.7
MOCKITO-12b	CCE	-	-			-			-		
MOCKITO-34b	AIOOBE	-	-			-			-		
MOCKITO-36b	NPE	1	15	10.9	6.9	15	9.2	7.5	15	13.7	11.3
MOCKITO-38b	NPE	-	-			-			-		
MOCKITO-3b	AIOOBE	-	-			-			-		
XRENDERING-418	NPE	-	-			-			-		
XWIKI-12482	NPE	-	-			-			-		
XWIKI-12584	CCE	-	-			-			-		
XWIKI-13031	CCE	3	15	25.8	17.4	15	47.2	67.0	10	249.0	175.0
XWIKI-13096	NPE	-	-			-			-		
XWIKI-13303	NPE	-	-			-			-		
XWIKI-13316	NPE	2	15	37.9	47.7	15	16.6	34.6	15	31.3	86.8
XWIKI-13377	CCE	1	15	10.7	8.6	15	11.8	7.7	15	4.8	3.9
XWIKI-13616	NPE	3	15	4.1	0.1	15	4.0	0.0	15	4.0	0.0
XWIKI-14227	NPE	-	-			-			-		
XWIKI-14319	NPE	1	15	87.0	21.2	15	89.4	17.5	15	87.8	15.2
XWIKI-14475	**NPE**	**1**	**15**	**117.1**	**53.6**	**-**			**-**		
XWIKI-13916	CCE	1	15	59.7	19.8	14	65.0	13.6	15	57.6	13.8
XWIKI-14612	NPE	1	15	8.9	2.0	15	8.7	1.8	15	8.5	2.4

Table 4.3: Comparing coverage rate and test generation time between the optimization approaches, for cases where both optimization approaches in each pair reproduces the crash. P-values for both Wilcoxon tests and odds ratios are reported. Effect sizes and p-values of the comparisons are in bold when the p-values are lower than 0.05.

			Multi-Weighted				Weighted-Simple			
Crash ID	**Exception**	**Fr.**	$\hat{A}_{12}$	p	OR	p	$\hat{A}_{12}$	p	OR	p
CHART-4b	NPE	6	0.3	0.30	0.0	1.0	**0.8**	**<0.01**	0.0	1.00
LANG-12b	AIOOBE	2	0.5	0.50	0.0	1.0	0.4	0.70	0.0	1.00
LANG-33b	NPE	1	**0.9**	**<0.01**	0.0	1.0	0.5	0.30	0.0	1.00
LANG-39b	NPE	2	**0.9**	**<0.01**	0.0	1.0	0.4	0.10	0.0	1.00
LANG-47b	NPE	1	**0.9**	**<0.01**	0.0	1.0	0.4	0.70	0.0	1.00
LANG-57b	NPE	1	0.6	0.20	1.7	0.6	0.5	0.60	0.3	0.40
MATH-100b	AIOOBE	1	**0.1**	**<0.01**	0.0	1.0	0.5	0.40	0.0	1.00
MATH-32b	CCE	2	**0.3**	**<0.01**	0.0	0.5	0.4	0.50	0.0	1.00
MATH-4b	NPE	3	**0.4**	**0.04**	1.0	1.0	0.4	0.70	1.0	1.00
MATH-70b	NPE	3	**0.8**	**<0.01**	0.0	1.0	0.5	0.10	0.0	1.00
MATH-81b	AIOOBE	6	0.5	0.60	4.8	0.3	0.5	0.50	0.0	0.09
MATH-98b	AIOOBE	1	**0.3**	**<0.01**	0.0	1.0	0.6	0.20	0.0	1.00
MOCKITO-36b	NPE	1	0.2	0.60	0.0	1.0	0.3	0.30	Inf	1.00
XWIKI-13031	CCE	3	**0.3**	**0.03**	Inf	1.0	**0.1**	**<0.01**	6.5	0.10
XWIKI-13316	NPE	2	0.6	0.09	0.0	1.0	0.6	0.10	0.0	1.00
XWIKI-13377	CCE	1	0.6	0.50	0.0	1.0	**0.7**	**0.01**	0.0	1.00
XWIKI-13616	NPE	3	**0.5**	**<0.01**	0.0	1.0	**0.5**	**<0.01**	0.0	1.00
XWIKI-14319	NPE	1	**0.4**	**<0.01**	0.0	1.0	0.5	0.70	0.0	1.00
XWIKI-13916	CCE	1	0.3	0.60	0.0	1.0	0.6	0.08	0.0	1.00
XWIKI-14612	NPE	1	0.5	0.40	0.0	1.0	0.4	0.70	0.0	1.00

5 The Significance of Bug Report Elements

Open source software projects often use issue repositories, where project contributors submit bug reports. Using these repositories, more bugs in software projects may be identified and fixed. However, the content and therefore quality of bug reports vary. In this study, we aim to understand the significance of different elements in bug reports. We interviewed 35 developers to gain insights into their perceptions on the importance of various contents in bug reports. To assess our findings, we surveyed 305 developers. The results show developers find it highly important that bug reports include crash description, reproducing steps or test cases, and stack traces. Software version, fix suggestions, code snippets, and attached contents have lower importance for software debugging. Furthermore, to evaluate the quality of currently available bug reports, we mined issue repositories of 250 most popular projects on Github. Statistical analysis on the mined issues shows that crash reproducing steps, stack traces, fix suggestions, and user contents, have statistically significant impact on bug resolution times, for ∼70%, ∼76%, ∼55%, and ∼33% of the projects. However, on average, over 70% of bug reports lack these elements.

5.1 Introduction

Open source software projects often maintain issue repositories to manage feature requests and bug reports. There are potential advantages to using open issue repositor-

ies [45]. Contributors of software projects provide their inputs and maintain focused conversations over them. As a result, more bugs in software projects may be identified and fixed [45].

Bug reports contain various types of information, including: software version, crash description, reproducing steps, reproducing test cases, crash stack traces, and fix suggestions. To make bug reports consistent, often default templates are provided in project repositories, where certain required or at least recommended fields are specified to be filled by the contributors. Yet, the content and therefore quality of bug reports vary [224]. Potential reasons for this issue include: data loss during a software crash, difficulty to find crash data in log files, and lack of sufficient technical experience [224].

If too little data is provided in bug reports, then understanding the problem, and therefore reproducing it is nontrivial and time-consuming. On the other hand, reproducing software crashes is a vital step in software debugging. Developers need to know how to reproduce the crashes to be able to confirm the fixes they deliver. Furthermore, low quality bug reports may demotivate developers and therefore take longer to be processed.

The following are examples of bug reports from various popular projects on Github [1–8]. These examples illustrate when a crash stack trace or reproducing test case are missing, developers respond by first asking the bug reporter to provide these elements. Figure 5.1 shows a bug report [1] as well as the responses to the bug report. As Figure 5.1a shows, the bug report includes various elements such as actual behavior, reproducing steps, versions of various components, etc. However, the bug report misses a crash stack trace. As Figure 5.1b shows, the developers explicitly ask for the crash stack trace. Since after one month, this information is not provided, the bug report is closed.

We aim to understand the significance of various information in bug reports for software debugging. To gain an in-depth understanding of developers' perceptions, we interviewed 35 developers. We used Grounded Theory [40] [111] techniques to analyse the interview results. To examine the findings from the interviews, we surveyed 305 developers. Our findings confirm that crash description, crash reproducing steps and test cases, and stack traces are of high importance for developers when debugging. On the other hand, developers find extra information that users may provide such as fix suggestions, code snippets, and links to user content, such as screenshots, of lower importance.

To gain insights on how often important elements are included in bug reports and their impact on bug resolution times, we developed the *IMaChecker* approach. *IMa-*

Checker receives Github repositories as input, then mines all issues posted in the input repository. Once the issues are downloaded, *IMaChecker* analyses the issues to check whether they are bug reports, and if they contain elements including: crash description, reproducing steps or test cases, stack traces, code snippets, links to user content, or fix suggestions.

To create a corpus of repositories for evaluation, we first selected five popular languages used in Github according to *The State of Octoverse* [16] [17], which are namely: Javascript, Python, Java, PHP, and Ruby. For each language, we selected 50 most popular repositories, resulting in 250 repositories in total, on Github.

To analyse the impact of various elements of bug reports on bug resolution times, we used the Wilcoxon-Mann Whitney test.

To study realistic projects and maintain statistical power, only those projects which provided at least 10 issues for both experimental and control groups, were analysed. Experimental groups contained issues which only included the element of interest in the bug report (e.g., the issue only included stack traces). Control groups contained issues which only included general description of the crash. The results confirm that reproducing steps, stack traces, fix suggestions, and user contents have statistically significant impact on bug resolution times, for **∼70%, ∼76%, ∼55%, and ∼33%** of the projects, respectively. For code snippets, representative projects were not found.

Furthermore, we used descriptive statistics to report the average percentages of bug reports that include different bug report elements. Despite our findings on important bug report elements and their impact on bug resolution times, on average, over ∼70% of bug reports lack all important elements.

The above results help to raise awareness of the significance of various contents in bug reports for software debugging. Developers can use this information to prepare better templates for bug reports, in which all important elements are explicitly asked for. Furthermore, future work may investigate means to support and enable users to find and provide the information elements.

The contributions of the paper[1] are the following:

1. an extensive report from developer interviews and surveys, in addition to the interview and survey questionnaires,
2. *IMaChecker* as an open source tool, written in Python, which can be used to mine and analyse issues from Github repositories, and

[1] The interview and survey questions, as well as the dataset package are available via the following DOI: 10.5281/zenodo.3666763

3. a reproducible package which contains the data set of all mined issues from 250 most popular Github repositories, together with the R scripts used to analyse the mined data.

The remainder of this chapter is organized as following: Section 5.2 presents the research methodology. Section 5.3 presents the *IMaChecker* approach. Section 5.4 presents the results. Section 5.5 provides discussion on the findings of the paper. Section 5.6 provides related work. Finally, Section 5.7 concludes the paper.

5.2 Research Methodology

The overarching goal of this study is to identify the significance of elements of bug reports for software debugging. Therefore we define the following research questions:

- **RQ_1:** What types of information do developers perceive as important in bug reports?

  Motivation: The quality of bug reports varies depending on the kinds of information which are included in them. The study by Zimmermann et al. [224] shows developers and user of Apache, Eclipse, and Mozilla find reproduction steps and crash stack traces to be the most useful elements in bug reports. However, there is little knowledge about the other elements in bug reports and the extent to which they are perceived as important for software debugging. We raise **RQ_1** to broaden our perspective and gain a holistic understanding about the extent to which different bug report elements are of importance for software debugging in developers' perception.

  Data collection and analysis: To answer **RQ_1** we aim to combine interviewing developers with surveying them. By conducting interviews, we intend to gain a preliminary understanding of developers' views on bug reports and the role that each bug report element plays in the process of software debugging. We use thematic analysis and coding techniques to analyse the interview data. Using the information from the interviews, we devise a survey study where we examine and quantify the results from the interviews. We use descriptive statistics to measure the percentages of participants who consider a bug report element as highly important, moderately important, slightly important, or not important for software debugging.

- **RQ_2:** Do the important elements in bug reports impact bug resolution times?

Motivation: While with **RQ1** we identify the extent to which different bug report elements are important in developers' perception, it would still be unclear what impact these elements may have on bug resolution times. Therefore, we raise **RQ2** to understand the effect of different bug report elements on the time it takes to resolve bug reports.

Data collection and analysis: To answer **RQ2**, we use Github APIs to mine bug report repositories from Github. Once we obtain the bug reports from Github, we use the *IMaChecker* technique (presented in Section 3) to parse the bug reports statically. Once the static analysis is done, we then use statistical tests to measure the impact of various bug report elements on bug resolution times.

- **RQ3:** How often do bug reports contain the important elements?

  Motivation: With **RQ1** and **RQ2**, we gain an understanding about the extent to which different bug report elements are important for bug resolution. However, it would still be unclear how often these important elements are actually provided in bug reports. For example, as the study by Zimmermann et al. [224] shows, elements such as crash stack traces are difficult to provide.

  Data collection and analysis: To answer **RQ3**, we use the results from the static analysis which is performed by *IMaChecker* on the mined bug reports. As a result of this analysis, different elements of bug reports are identified. Therefore, we use descriptive statistics to report how often various elements appear in bug reports.

By combining qualitative and quantitative research methods, we use a mixed-method research approach [89] to answer the research questions. In what follows, we further present the research techniques we used.

5.2.1 Interviews

To answer **RQ1**, we followed a qualitative research method [89]. We interviewed 35 developers in order to gain an understanding of their debugging techniques and the kind of information they find important to receive in bug reports. In what follows, we present the interview protocol, the participants, and data analysis technique we used for the interviews.

5.2.1.1 Protocol

We conducted semi-structured interviews [125], in which we combined broad and open-ended questions[2] with specific questions. In this way, we let participants freely respond and explore relevant topics, while we made sure the intended topics were also explored by asking specific questions. As suggested by Barriball et al. [62] [126], we conducted four pilot interviews before we performed the main interviews. As a result, we received feedback on the general flow of the questions from two of the pilot interviews. According to this feedback, we should have noted the role the participants play in their organization. Therefore, we added two questions in the interview instrument where we specifically ask about the role of the participant and we ask if the participant can briefly explain what this role entails.

We let the participants know in advance that we intend to use the data anonymously. Prior to the interviews we got permission from the participants to record the interviews. Furthermore, 15 out of 35 interviews were conducted through online calls because the developers were not available in person. Each interview took between 20 minutes to 60 minutes.

5.2.1.2 Participants

We intended to form a diverse group of participants. Thus, using our social contacts, we reached out to developers who work in the following areas: e-commerce development, ERP application development, automotive industry, artificial intelligence, embedded programming, and database administrating. We sent personalized emails to 50 developers who worked in these industries. 40 people with background in e-commerce development, ERP application development, and automotive industry agreed to participate in this study. After 35 interviews we reached theoretical saturation [111]. Figure 5.2 shows the years of professional experience of the interview participants. The participants had at least five and at most 25 years of professional experience as a developer.

5.2.1.3 Data Analysis

After the interviews, we manually transcribed the recorded interviews. To analyse the collected data, we used thematic analysis [110] [71] to identify emerging categories

[2] The interview questions are provided in the reproduction package, via DOI: 10.5281/zenodo.3666763

in the transcripts. Thematic analysis is a technique that is used when analysing textual data. Using this technique, the first author read the transcripts intensively. The first author then used open and axial coding techniques [164] to tag the pieces of text which would relate to $\mathbf{RQ_1}$. After identifying the tags, the first author reviewed them and grouped them together to form more generic themes. Ultimately, the identified themes addressed two main categories: the debugging techniques developers used, and the kind of information in bug reports they considered important for software debugging. Figure 5.3 is a visual representation of the main themes that were identified throughout this process.

5.2.2 Surveying Developers

To generalize the findings from the interviews, and measure the prevalence of the debugging practices and developers' perceptions on the importance of different bug report elements for software debugging, we surveyed 305 developers. In what follows, we describe the survey protocol, survey participants, and our data analysis approach.

5.2.2.1 protocol

To construct the survey[3], we used guidelines from Fink [100], De Vaus [91], Pfleeger and Kitchenham [183], and Kitchenham and Pfleeger [139]. We used closed questions to make the survey more compelling for the participants to fill in. To avoid forcing the participants to choose an option, for each closed question, there was an option where the participants could write their responses. We provided a brief overview of the purpose of the survey in the introduction. We let participants know we would use the data anonymously.

Before sending out the survey, we used pilot studies with four participants who were professional developers. We asked the participants to fill in the survey, and provide us with their feedback about the structure and questions of the survey. One feedback we received was about the length of the introduction at the beginning of the survey. The participant mentioned that the introduction could be shortened for more readability. In addition, another feedback was about asking the participants if they wish to receive the results after the survey is done. This is why we added one last question at the end of survey where the participants can leave their contact information if they wish to receive the results. We discarded the results of the pilot studies from the main results in this paper.

[3]The survey questions are provided in the reproduction package, via DOI: 10.5281/zenodo.3666763

5.2.2.2 Participants

To find participants for the survey, we searched for trending developers[4]. In addition, we searched for active developers from 85 popular software projects on Github. The main rationale behind this approach for selecting the participants is that we intended to involve participants who are selected from a pool of experienced developers. From each project we selected three to four active developers. This way we reached out to 317 people. We sent personalized emails to these developers, and briefly explained the purpose of the study to them. We received 222 responses. In addition, we used the snowballing technique [167] to collect more participants. After the participants responded to the survey, we asked them if they could introduce us to colleagues who would be interested to participate in the study. We sent personalized emails to 105 developers, and we received 83 responses. In total, we received 305 responses for the survey. Figure 5.4 shows the years of professional experience of the survey participants.

5.2.2.3 Data Analysis

To analyse the results of the survey, we used descriptive statistics to report the findings from the closed questions. Therefore, for each bug report element, we simply measure the percentages of participants who perceive the element as highly important, moderately important, slightly important, or not important. Furthermore, we count the number of participants who are project manager, software developer, software tester, software maintainer, scrum master, or those who indicate any other type of role they play. We also count the number of years of professional experience the participants indicate to have. For the questions which let the participants write an answer in text, we use thematic textual analysis to identify emerging categories from the written texts.

5.2.3 Mining Github Issues

To answer **RQ**2 and **RQ**3, we mined and analysed issues from 250 projects on Github. To do so, we developed the Issue Miner and Checker (*IMaChecker*) approach. *IMaChecker* mines the issues of the received repositories, and further checks them to detect whether stack traces, reproducing steps, fix suggestions, code snippets, and user content are provided in the issues. In Section 5.4 we will further describe the *IMaChecker* approach.

[4]Through https://github.com/trending/developers

To select the projects, we first identified the five most popular programming languages used in Github. According to *The State of Octoverse* [16] [17], the languages are: Javascript, Python, Java, PhP, and Ruby. Next, based on the measures of popularity that Borgens et al. identify [68], for each language, we selected 50 projects, 250 projects in total, that have the most number of stars and forks. Table 4 (in Appendix A) presents an overview of the projects, the number of stars, forks, contributors, as well as the year in which the first commits were provided in the project[5].

5.2.3.1 Analysis of the Mined Issues

To measure the impact of various elements of bug reports on bug resolution times, we use the Wilcoxon-Mann Whitney statistical test. This is a non-parametric test that is used to analyse the impact of an independent variable that is at least ordinal. When it is not possible to make assumptions about whether the data is normally distributed or not, Wilcoxon-Mann Whitney is an alternative approach that can be used instead of techniques such as the independent samples t-test. Since in this case the dependent variable is resolution time, we only consider closed issues where the reported bug is fixed. The null hypotheses in these experiments are the following:

- **H_{01}:** the time it takes to close a bug report which only includes a problem description and crash stack trace is the same as the time it takes to close a bug report that only includes a problem description.
- **H_{02}:** the time it takes to close a bug report which only includes a problem description and reproduction steps is the same as the time it takes to close a bug report that only includes a problem description.
- **H_{03}:** the time it takes to close a bug report which only includes a problem description and fix suggestion is the same as the time it takes to close a bug report that only includes a problem description.
- **H_{04}:** the time it takes to close a bug report which only includes a problem description and user content is the same as the time it takes to close a bug report that only includes a problem description.
- **H_{05}:** the time it takes to close a bug report which only includes a problem description and code snippet is the same as the time it takes to close a bug report that only includes a problem description.

We use experimental and control groups. In experimental groups, only those issues are present which only include one of the bug report elements e.g., stack traces,

[5] The results were collected on 2019-05-15.

depending on the element under analysis. Control groups contain those issues in which none of the bug report elements are present. To analyse realistic projects and maintain statistical power, we make sure that the sample sizes are at least 10, i.e., at least 10 issues are analysed in each group. Furthermore, the test does not assume that the samples are normally distributed. We consider α=0.05 for Type I errors to assess the significance of the results.

We use the Vargha-Delaney $\hat{A}_{12}$ statistic [207] to assess the effect sizes. Vargha-Delaney $\hat{A}_{12}$ is also a non-parametric approach for comparing performances of two independent groups. The outcome of this test is a value between 0 and 1. Therefore, if the outcome of Vargha-Delaney $\hat{A}_{12}$ is 0.5, the two groups perform the same. On the other hand, if the result of Vargha-Delaney $\hat{A}_{12}$ is less than 0.5, the first group performs worse, while if the outcome is larger than 0.5, the first group perform better than the second group. Using Vargha-Delaney $\hat{A}_{12}$, we report effect magnitudes which indicate the following effect sizes: negligible, small, medium, and large.

5.3 The *IMaChecker* Approach

To mine and analyse the issues, we developed the Issue Miner and Checker (*IMaChecker*) in Python 3. This approach has been tested on a Linux kernel version 4.15, as well as a MacOS 10.14 machine.

Figure 5.5 presents an overview of the approach. *IMaChecker* receives the list of Github Repositories as input. Next, *IMaChecker* downloads all issues posted to the repository, using the Github API [9]. After the issues of all projects are downloaded, the user can use the APIs that *IMaChecker* provides to analyse the downloaded issues.

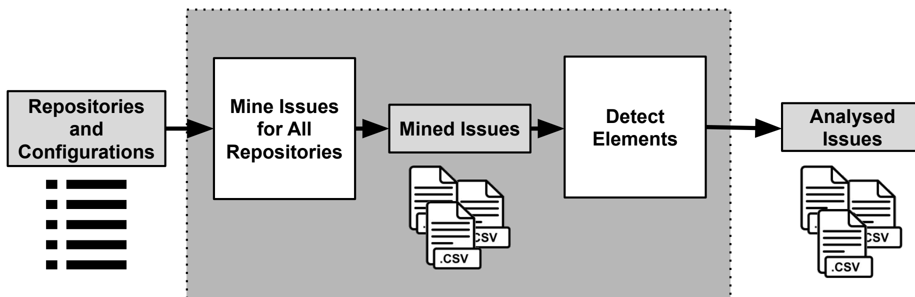


Figure 5.5: The figure presents an overview of the *IMaChecker* Approach.

IMaChecker uses regular expressions to detect issues that are originally labeled as

bugs. Often various terms (e.g. “crash") are used to mark an issue as a bug in issue repositories. Therefore, it is possible to feed *IMaChecker* with specific terms of interest to detect originally labeled bugs.

IMaChecker uses specific strings and regular expressions to detect whether the issues include stack traces, reproducing steps, fix suggestions, code snippets, and links to user contents. To identify the strings and design the regular expressions, we studied 255 bug reports which were randomly selected from the projects presented in Table 4 (in Appendix A). After we reached the saturation point and did not find any new keys in the context of the bug reports, we collected a pool of strings which were commonly used to refer to different bug report elements. Table 2 shows these strings and regular expressions.

Table 5.1: The strings and regular expressions we used to parse reproduction steps, fix suggestions, user contents, and code snippets in bug reports.

Element	**Strings or regular expressions**
Reproduction Step	“reproducing steps" “steps to reproduce" “reproduce" “reproducible test case" “reproducible" “to reproduce" “minimal reproduction"
Fix Suggestion	“fix suggestion" “suggestion to fix" “suggestions to fix" “suggest" “suggestions"
User Content	https://user-images.githubusercontent.com/[\Sa-z0-9A-Z]+.[a-zA-z]
Code Snippet	[\w+\s]+```` \\r \\n

Since each programming language uses a specific format to generate stack traces, *IMaChecker* uses five different regular expressions that are adjusted to the five different stack trace formats in Javascript, Python, Java, PhP, and Ruby. Table 2 shows examples of stack traces for different languages as well as the regular expressions used to detect them.

If *IMaChecker* detects a stack trace in the issue, the exception type of the stack trace is recorded as well. This can be used when one wishes to report frequency of various exception types. In addition, if *IMaChecker* detects crash reproducing steps or stack traces, or fix suggestions, then it automatically marks the issue as a bug. This can be

Table 5.2: Examples of stack traces in different languages as well as the regular expressions used to detect them.

Language	Example	Regular Expression
Javascript	at split (angular.js:27114) at updateClasses (angular.js:27043) ..., from [10]	[\s]+at[\s]+[\w+.]+[\s]+\([/*\ w+]+.js:[0-9]+:*[0-9]*\)\\r\\n
Python	Traceback (most recent call last): File “facedetect.py", line 251, in <module> main_loop() ..., from [11]	Traceback\s\Smost\srecent\scall\s last\S: |File[\s].+,[\s]+line[\s] +[0-9]+,[\s]+in[\s]+.+\\r\\n
Java	at android.view.ViewGroup. dispatchDraw(ViewGroup.java:3554) at android.view.View.updateDisplay ListIfDirty(View.java:15237) ..., from [12]	[\s]+at[\s]+[\w+.\S]+\(\w+.java :[0-9]+\)
PHP	#0 /Applications/MAMP/htdocs/ learning/laravel/larabootstrap5/ vendor/laravel/framework/src/ Illuminate/Foundation/Bootstrap/ HandleExceptions.php(118): ..., from [13]	\#[0-9]+\s+[\w+\S]+.php \([0-9]+\):
Ruby	/home/navin/.rvm/gems/ ruby-2.2.1/gems/sprockets-3.4.0/lib/ sprockets/sass_processor.rb:278:in sprockets_context ..., from [14]	[\w+\S]+.rb:[0-9]+:in\s+

useful as not always the issues are labeled in a Github Repository.

Furthermore, Figure 5.6 shows an example[6] of a bug report from the AngularJS project. As the example shows, this bug report contains a description of a memory allocation problem together with a snapshot that is included as a .png file. When IMaChecker parses the bug report content, it detects the user content is provided through the .png file.

[6]This bug report can be found via: https://github.com/angular/angular.js/issues/16853

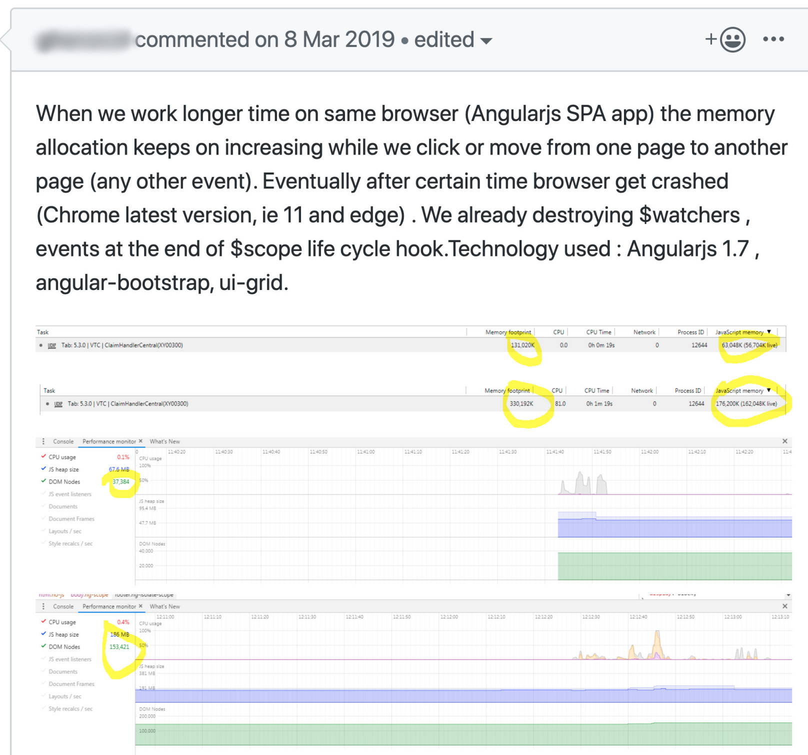


Figure 5.6: An example of a bug report from the AngularJS project.

To evaluate the precision of the *IMaChecker* approach, we randomly selected 100 bug reports from the projects in Table 4 (in Appendix A). We manually analyzed the bug reports and made an account of the elements included in them. We then ran *IMaChecker* in order to detect the bug report elements automatically. The precision was around 92%. This was because there were bug reports in which reproduction steps or stack traces were provided through user contents (e.g., through links to external pages). Therefore, it was not possible for the *IMaChecker* approach to detect these elements by parsing the texts.

5.4 Results

We used a mixed-method research approach to discover the significance of bug report elements in software debugging. To answer the research questions, we combined interviewing developers with surveying them. In addition, we mined 250 issue repos-

itories and used descriptive statistics as well as statistical tests on the mined issues. In this section, we present the results and thereby answer $\mathbf{RQ_1}$, $\mathbf{RQ_2}$, and $\mathbf{RQ_3}$.

5.4.1 RQ_1. What types of information do developers perceive as important in bug reports?

During the interviews, in order to get a broad understanding of the debugging process the developers have, we asked the participants to describe the debugging approach they take typically. In this regard, we gained the following insights. The interview participants often prefer using *printf*s for debugging. When a crash is complex, then 45% of the interview participants indicated they would use a debugger to further analyse the execution scenarios. In addition, all participants indicated that especially when they face a new error they have not seen before, they typically google the error message. Often it is the case that on platforms such as *stackoverflow*[7], someone else has posted a similar problem, which provides the participants an opportunity to get further insights. Otherwise, they may open a new issue on those platforms, share their problem, and ask the community to look into the questions.

To answer RQ_1, we derived 7 categories from the interview results which indicate the information elements that developers perceive as important, which they prefer to be included in bug reports: crash description, software version, reproduction steps, stack traces, code snippets, user content and fix suggestions. To quantify these results and gain insights into the extent to which these elements are of importance for debugging, we surveyed the developers.

Figure 5.7 presents the results from the surveys. According to the results, 96% of the participants find reproduction steps or test cases of high importance while 4% of them believe reproducing steps or tests are moderately important. 95% of the participants find crash stack traces of high importance while 5% of them find crash stack traces of moderate importance.

In addition, around 89% of the participants find crash description of high importance, while 11% of them believe crash descriptions are of average importance. Around 12% of the participants find software version of high importance, while 66% of them believe software versions are of average importance.

Around 14% of the participants find code snippets of high importance, while 68% of them believe code snippets are of average importance. 16% of the participants find code snippets of slight importance. 2% of the participants find code snippets of no

[7] https://stackoverflow.com/

importance for software debugging. In this regard, a participant mentioned: “I prefer to receive them in a pull request not in a bug report."

13% of the participants find software versions of slight importance. 9% of the participants do not find software version important for software debugging. One of the participants indicated: “Often the version is understood from the context of the bug report. For example, certain features are only available in our latest release."

Around 8% of the participants find fix suggestions of high importance, while around 81% of them believe fix sugestions are of average importance. 11% of the participants believe fix suggestions are of little importance.

Around 3% of the participants find user contents of high importance, while 74% of them believe user contents are of average importance. 19% of the participants find user contents of slight importance. 3% of the participants find user contents of no importance for software debugging. In this regard, a participant mentioned: “User content could be anything. They are supplementary."

5.4.2 RQ_2. Do the important elements in bug reports impact bug resolution times?

Table 3 presents the results of Wilcoxon-Mann Whitney and Vargha Delaney $\hat{A}_{12}$ statistical analysis on four elements of bug reports, namely: stack traces, crash reproducing steps or test cases, fix suggestions, and user contents.

Since we compare resolution times, we only consider closed issues where the reported bug is fixed. To maintain statistical power, we made sure that in each project, there are at least 10 issues which have none of the comparison elements in the description (control group), and there are at least 10 issues which have only the comparison factor (e.g., stack traces) in the description (experimental group). If a project does not provide such groups, we excluded it from the analysis.

To analyse the impact of stack traces, we found 139 projects, which provide the control and experimental groups. In 106 projects out of 139 projects (∼76%) statistically significant results show that including stack traces impacts the bug resolution times. For 33 projects (∼24%) no conclusion was drawn.

To analyse the impact of reproducing steps or test cases, we found 142 projects, which provide the control and experimental groups. In 100 projects out of 142 projects

(∼70%) statistically significant results show that including reproducing steps impacts the bug resolution times. For 42 projects (∼30%) no conclusion was drawn.

To analyse the impact of fix suggestions, we found 148 projects, which provide the control and experimental groups. In 81 projects out of 148 projects (∼55%) statistically significant results show that including fix suggestions impacts the bug resolution times. For 67 projects (∼45%) no conclusion was drawn.

To analyse the impact of user contents, we found 33 projects, which provide the control and experimental groups. In 11 projects out of 33 projects (∼33%) statistically significant results show that including user contents impacts the bug resolution times. For 22 projects (∼67%) no conclusion was drawn.

Table 5.3: The table shows the results from the Wilcoxon-Mann Whitney, and Vargha Delaney $\hat{A}_{12}$ statistical analysis on four elements of bug reports, namely: Stack Traces, crash Reproducing Steps, Fix Suggestions, and User Contents. **p** indicates the p values from the Wilcoxon test. **v-mag.** indicates the Vargha Delaney measures of magnitude, which show the effect sizes. **l**,**m**,**s**, and **n**, indicate large, medium, small, and negligible effect sizes, respectively. **-** indicates that control or experimental groups were not found for the comparison factor.

	Stack Trace		Reproducing Step		Fix Suggestion		User Content	
Repository	p	v-mag.	p	v-mag.	p	v-mag.	p	v-mag.
30-seconds/30-seconds-of-code	-	-	0.094	m	0.561	n	-	-
activeadmin/activeadmin	0	l	0	l	0	m	-	-
adobe/brackets	-	-	0	m	0	s	-	-
angular/angular.js	-	-	0	m	0	s	-	-
ansible/ansible	0	m	0	n	0.001	n	0.015	m
apache/incubator-dubbo	0	m	0.005	s	0.125	s	-	-
apache/incubator-echarts	-	-	0	m	0.485	n	0.745	n
apache/incubator-zipkin	0.09	s	0.146	s	0.026	m	0.499	n
atech/postal	0.517	s	-	-	-	-	-	-
atom/atom	0	l	0	m	0	s	-	-
axios/axios	0.048	s	0.088	s	0.714	n	-	-
babel/babel	0	s	0.02	n	0	s	0.002	m
bazelbuild/bazel	0.002	s	0.002	n	0.059	n	-	-

bcit-ci/CodeIgniter	-	-	0.701	n	0.281	n	-	-
BetterErrors/better_errors	0.209	s	-	-	-	-	-	-
briannesbitt/Carbon	-	-	0.025	m	0.503	s	-	-
bumptech/glide	0.02	n	0.17	n	0.046	s	-	-
CachetHQ/Cachet	0	s	0	s	0.065	s	0	l
cakephp/cakephp	0.003	m	0.047	s	0.389	n	-	-
capistrano/capistrano	0	l	0	l	0.001	m	-	-
carrierwaveuploader/carrierwave	0	l	0	l	0	l	-	-
celery/celery	0	m	0	s	0.001	s	-	-
certbot/certbot	0	s	0.026	s	0.046	n	-	-
chartjs/Chart.js	-	-	0	s	0.001	s	0.172	n
chrisbanes/PhotoView	0	l	-	-	-	-	-	-
composer/composer	0.002	m	0	s	0	s	-	-
deeplearning4j/deeplearning4j	0	s	0.412	n	0.979	n	0.507	n
deployphp/deployer	-	-	0	s	0.357	n	-	-
diaspora/diaspora	0.081	n	0.014	n	0.009	s	-	-
dingo/api	0	l	0.009	m	0	l	-	-
docker/compose	0.005	n	0	s	0.697	n	-	-
Dogfalo/materialize	0	l	0	l	0	l	-	-
elastic/elasticsearch	0.786	n	0	n	0	s	-	-
elastic/logstash	0.001	n	0	s	0.031	n	-	-
encode/django-rest-framework	0	m	0	m	0	s	-	-

explosion/spaCy	0.002	s	0.025	n	0.945	n	-	-
expressjs/express	0.006	s	0.003	s	0.036	s	-	-
facebook/create-react-app	0	m	0	s	0.615	n	-	-
facebook/fresco	0	m	0	m	0.025	s	0.015	m
facebook/react	0	l	0	l	0	s	-	-
facebook/react-native	0	m	0.969	n	0.12	n	0.051	s
facebook/stetho	0.77	n	-	-	-	-	-	-
fastlane/fastlane	0	s	0.898	n	0	s	0.478	n
fluent/fluentd	0.001	s	0.04	s	0.193	s	-	-
FortAwesome/Font-Awesome	-	-	0	m	0	s	0.546	n
freeCodeCamp/devdocs	0.523	n	0.124	s	0.006	l	-	-
freeCodeCamp/freeCodeCamp	-	-	0	l	0.072	n	0.7	n
FriendsOfPHP/PHP-CS-Fixer	-	-	0.681	n	0.117	s	-	-
gatsbyjs/gatsby	0.007	s	0	s	0.649	n	0.218	s
getgrav/grav	0.057	s	0.602	n	0.178	n	-	-
getredash/redash	0	m	0.037	n	0.021	m	0.158	s
getsentry/sentry	0.094	s	0.968	n	0.01	s	-	-
github/linguist	0.071	m	-	-	0.327	s	-	-
gollum/gollum	0	m	0.084	s	0.059	s	-	-
google/ExoPlayer	0	m	0	s	0	s	0.029	m
GoogleChrome/puppeteer	0	m	0.001	s	0.394	n	-	-
greenrobot/greenDAO	0.405	n	-	-	-	-	-	-
gulpjs/gulp	0	l	0	l	0	l	-	-
guzzle/guzzle	0.005	l	0.092	s	0.318	s	-	-
h5bp/html5-boilerplate	-	-	0.083	m	0.028	s	-	-
hakimel/reveal.js	-	-	0.115	s	0.407	n	-	-
hashicorp/vagrant	0	l	0	s	0	s	-	-
HelloZeroNet/ZeroNet	0.146	s	0.313	n	0.312	s	-	-

home-assistant/home-assistant	0.007	s	0.925	n	0.595	n	-	-
Homebrew/brew	0	l	0	m	0	m	-	-
huge-success/sanic	0.054	s	-	-	0.752	n	-	-
huginn/huginn	0.046	m	-	-	0.567	n	-	-
imathis/octopress	0	m	0.735	n	0.096	s	-	-
ipython/ipython	0	m	0	s	0.043	n	-	-
jakubroztocil/httpie	0.02	s	-	-	-	-	-	-
javan/whenever	0	l	-	-	0.001	l	-	-
jordansissel/fpm	0.116	s	-	-	-	-	-	-
jquery/jquery	-	-	0	l	0	l	-	-
kaminari/kaminari	0	l	0.108	s	0.236	s	-	-
kennethreitz/requests	0	l	0	l	0	l	-	-
keras-team/keras	0.048	s	0.02	s	0.005	m	-	-
Konloch/bytecode-viewer	0.865	n	-	-	-	-	-	-
laravel/framework	0	l	0	l	0	l	-	-
localstack/localstack	0.251	n	0.529	n	-	-	-	-
lodash/lodash	0.058	s	0.021	s	0.01	s	-	-
magento/magento2	0	m	0	s	0	m	-	-
matomo-org/matomo	0.699	n	0.244	n	0	s	0.28	n
meteor/meteor	0	l	0	s	0	s	-	-
Microsoft/vscode	0	s	0	n	0.889	n	0.007	n
middleman/middleman	0	l	0.001	m	0.09	s	-	-
mikepenz/MaterialDrawer	0.541	n	0.299	n	0.054	s	-	-

mitmproxy/mitmproxy	0	m	0.002	s	0.563	n	-	-
mockery/mockery	-	-	0.003	l	0.026	m	-	-
moment/moment	-	-	0	m	0.005	s	-	-
monicahq/monica	0.595	n	0.192	s	-	-	0.011	s
mrdoob/three.js	0.002	m	0.002	s	0.005	n	0.696	n
mui-org/material-ui	0	l	0	m	0	m	0	m
mybatis/mybatis-3	0.615	n	0.196	n	0.876	n	-	-
NationalSecurityAgency/ghidra	-	-	0.731	n	-	-	0.819	n
netty/netty	0.002	s	0	s	0	m	-	-
nextcloud/server	0.458	n	0.549	n	0.033	n	0.505	n
nicolargo/glances	0.333	n	-	-	0.998	n	-	-
nodejs/node	0	m	0	s	0.049	n	-	-
nostra13/Android-Universal-Image-Loader	0.001	m	-	-	0.005	m	-	-
octobercms/october	0	l	0	s	0	m	0.057	s
omniauth/omniauth	0.01	l	-	-	0.406	s	-	-
pallets/flask	0	l	0.025	m	0.021	m	-	-
pandas-dev/pandas	0	s	0	s	0.512	n	-	-
parcel-bundler/parcel	0	m	0	s	0.029	s	0.896	n
phalcon/cphalcon	0	s	0	s	0.528	n	-	-
phanan/koel	-	-	0.038	s	0.014	m	-	-
PhilJay/MPAndroidChart	0.365	n	0.226	s	0.24	n	-	-
plataformatec/devise	0	l	0	m	0	l	-	-
plataformatec/simple_form	0	l	0.018	m	0	l	-	-

prettier/prettier	0	l	0.005	s	0.674	n	0.271	s
pypa/pipenv	0	l	0	l	0	m	-	-
rapid7/metasploit -framework	0	s	0	m	0.428	n	-	-
react-native-community/lottie -react-native	-	-	0.06	m	-	-	-	-
ReactiveX/Rx Java	0.637	n	0	m	0.042	s	-	-
ReactTraining/ react-router	0	l	0	l	0	l	-	-
realm/realm-java	0	s	0	s	0.01	s	-	-
reduxjs/redux	-	-	0.002	l	0.203	s	-	-
resque/resque	0	l	-	-	0.007	m	-	-
roots/sage	-	-	0	l	0.01	m	-	-
rubocop-hq/ru bocop	0	m	0.447	n	0.23	n	-	-
ruby-grape/grape	0	s	-	-	0.525	n	-	-
scikit-learn/ scikit-learn	0	s	0	m	0.2	n	-	-
scrapy/scrapy	0	m	0.02	m	0.013	s	-	-
sebastianberg mann/phpunit	0.167	n	0.491	n	0.305	n	-	-
Seldaek/mono log	0.002	l	-	-	0.004	l	-	-
Semantic-Org/ Semantic-UI	0.577	n	0.99	n	0.149	n	0.008	m
serverless/ser verless	0.001	s	0.346	n	0.089	n	0.635	n
sferik/rails_ admin	0.005	s	0.457	n	0.954	n	-	-
Shopify/liquid	0.001	l	-	-	-	-	-	-
signalapp/ Signal-Android	0	l	0	l	0	l	-	-
sinatra/sinatra	0.001	m	0.189	s	0.407	n	-	-
skylot/jadx	0.191	s	-	-	-	-	-	-
slimphp/Slim	0.992	n	0.478	n	0.12	s	-	-

socketio/socket .io	-	-	0.001	s	0.018	s	-	-
spring-projects/ spring-boot	0	n	0	s	0.069	n	-	-
spring-projects/ spring-framework	0	n	0	s	0	s	-	-
sqlmapproject/ sqlmap	0.359	n	0.054	n	0.031	s	-	-
square/okhttp	0	m	0	m	0.002	s	-	-
square/retrofit	0.001	l	0.131	m	0.057	m	-	-
StevenBlack/ hosts	0.93	n	-	-	0.467	s	-	-
storybooks/ storybook	0.037	s	0	s	0.507	n	0.67	n
stympy/faker	0.002	l	-	-	-	-	-	-
symfony/symfony	0	m	0	s	0.587	n	0.034	s
teamcapybara/ capybara	0	l	0	l	0	l	-	-
Tencent/tinker	0	l	-	-	-	-	-	-
tensorflow/models	0	m	0	l	0.001	m	-	-
the-control-group/ voyager	-	-	0	s	0.372	n	0.652	n
thepracticaldev/ dev.to	-	-	0	l	0.763	n	0.049	s
thoughtbot/bour bon	-	-	0	l	0.001	l	-	-
thoughtbot/ factory_bot	0.01	m	0.325	s	0.317	s	-	-
thoughtbot/pap erclip	0	l	0	l	0.005	s	-	-
tmuxinator/ tmuxinator	0.006	m	0.827	n	-	-	-	-
tootsuite/masto don	0	s	0	m	0.622	n	0.902	n
trailofbits/algo	-	-	0.007	s	0.873	n	-	-
TryGhost/Ghost	0	l	0	s	0.001	s	0.515	n
twbs/bootstrap	0.009	m	0	s	0	m	-	-

twbs/bootstrap-sass	0.626	n	0.909	n	0.067	s	-	-
vuejs/vue	0	l	0	l	0	m	-	-
webpack/webpack	0	l	0	l	0	s	-	-
wix/react-native-navigation	0.707	n	0.825	n	1	n	-	-
yarnpkg/yarn	0.016	s	0.029	n	0.429	n	-	-
yiisoft/yii2	0	s	0	s	0	n	-	-
ytdl-org/youtube-dl	0	m	0	m	0	l	-	-
zeit/next.js	0	m	0	s	0.067	n	-	-
zxing/zxing	0.001	l	0	l	0	l	-	-

5.4.3 RQ3. How often do bug reports contain the important elements?

To identify how often various bug report elements are included in bug reports, we used *IMaChecker*[8] to mine and analyse issue repositories from 250 Github projects. In total, 835381 issues were mined, out of which 89761 issues (∼11%) were open while 745620 issues (∼89%) were closed. 114053 bug reports (∼29.64%) were originally labeled as bugs in bug repositories while 219803 bug reports (∼70.36%) were automatically detected.

According to the results, for 228 projects, crash reproducing steps and stack traces were detected. For 244 projects fix suggestions were detected. For 226 projects user contents were detected. For 178 projects code snippets were identified. Finally, for 34 projects no bugs were originally labeled while *IMaChecker* detected bugs.

For *kilimchoi/engineering-blogs*, *doctrine/inflector*, and *doctrine/lexer* repositories no issues were originally or automatically marked as bugs. These repositories have 66, 27, and 2 issues, respectively. For these repositories, no reproducing steps, stack traces, fix suggestions, code snippets, or user contents were detected. For more detailed results, please see Table 5 in Appendix B.

[8]The mining was done on 2019-05-13.

In addition, Figure 5.8 presents the average percentages of different bug report elements. According to Figure 5.8, on average, ∼27.16% of the bug reports included stack traces, ∼27.07% of the bug reports included reproducing steps, and ∼20.59% of the bug reports included fix suggestions. In addition, on average, ∼14.23% of the bug reports included user contents, and ∼1.06% of the bug reports included code snippets.

5.5 Discussion

In this paper, we aim to identify the contents in bug reports that are of importance for debugging. Therefore, we sought for developers' perceptions in this regard, we analysed whether any of the bug report elements impact bug resolution times, and we measured how often various information elements are included in bug reports.

Our results show that certain elements, namely: crash description, reproducing steps, and stack traces are of high importance for debugging in developers' perceptions. According to the statistical analysis, reproducing steps, stack traces, fix suggestions, and user contents have statistically significant impacts on bug resolution times. Despite the above findings, as Figure 5.8 shows, on average, over ∼ 70% of the bug reports lack these elements. In what follows we further discuss the findings.

5.5.1 Bug Report Templates and User Support

In order to keep the issues consistent, and make sure certain elements are provided in bug reports, repositories often provide templates for reporting issues. The specified elements in such templates vary. While these templates often specify reproducing steps, or fix suggestions as fields to be filled by the users, stack traces, user contents or code snippets are not mentioned in the templates. Therefore, it is up to the issue reporter to provide them.

Our results show each of those elements, particularly stack traces, impact the bug resolution times. Therefore, to help keep the structure of issues consistent, and make sure important elements of bug reports are asked for, it is important to provide complete and well-structured bug report templates. The results presented in this paper help increase awareness in this regard.

On the other hand, as Zimmermann et al. [224] report, it may not be possible for users to provide certain information in their bug reports while at the same time it is important to do so. It is simply because important information are not always easy to be found. For example, stack traces are often hidden in log files, and therefore, it is not easy to find them, even if the issue templates ask for them. Therefore, future work may investigate means to support users and enable them to provide important information in bug reports.

5.5.2 Representative Samples

When analysing the impact of various bug report elements, many projects were excluded from the analysis because they did not offer representative samples for experiment and control groups. This is why it was not possible to analyse the impact of code snippets on bug resolution times.

The automated mechanism in *IMaChecker* helps increase the number of bug reports, thereby the sample sizes for experimental groups. *IMaChecker* detects whether an issue is a potential bug if a certain element such as stack trace or fix suggestion is included in the reported issue.

However, if an issue does not include any of the elements, the only way to identify whether it is a bug report would be to check the labels put on the issue. At the same time, many of the bug reports were not originally labeled as bugs. Therefore, they could not be used in the control groups. As a result, many projects were excluded from the analysis.

This observation highlights the importance of properly documenting the bug reports. The *IMaChecker* approach provides a more accurate overview of the issues if bug reports are properly marked by developers.

5.5.3 Internal Validity of the Experiments

Internal validity of a study refers to how well the findings of the study explain a claim about a cause and effect. In the context of our study, threats to internal validity refer to alternative reasons why a bug report is closed more quickly than others.

In some cases, bug reports are created, however they either have no content or very minimal amount of information. We have observed that these kinds of bug reports are typically very quickly closed because there is not much that can be done for them. When developers close such bug reports, often they ask the contributors who opened

the bug reports to provide further information. Furthermore, sometimes bug reports are re-opened. One possible explanation is that the issue, which was addressed previously, resurfaces, either for the same contributor who previously opened the issue or someone else.

In our experiments, *IMaChecker* automatically checks the contents in experimental groups and control groups before they are included in the statistical tests. Therefore, the bug reports used in these experiments are never entirely empty. However, it could be that they are closed because they included too little information. In addition, in these experiments, we do not check whether an issue is re-opened later on. This is mainly due to the fact that the information that can be retrieved through the Github API does not include sufficient details with regards to whether the issue was re-opened or not.

5.5.4 Generalizability of Results

As Basili et al. [63] discuss, carrying out empirical work in software engineering is complex and time consuming. They argue that one reason for such complexity is that there are a large number of context variables. Therefore, creating a cohesive understanding of the experimental results requires effort.

We selected participants from three different industries, e-commerce, ERP, and automotive. In addition, the survey participants were either trending developers on Github or selected from over 85 distinct popular software projects. The professional experience of these participants ranged from one year to 27 years. While we intended to involve experienced developers in the survey, we did not ensure if the developers have experience in developing closed source projects or not.

To make a corpus of open-source projects, we selected 250 projects from Github. Github is a popular platform where over 2 million organizations and 96 million repositories are hosted to which over 31 million developers contribute, according to *The State of Octoverse* [15]. To select the open source projects, first we chose five popular programming languages, and then we used common measures of popularity, i.e., number of stars and forks, to identify the projects. Furthermore, we used statistical tests to analyse the results.

However, we can not claim that the findings are transferable to closed-source projects. Communication with users and debugging practices differ in closed-source projects. Future work may investigate closed-source projects as well as expert developers in the field, and compare the results with the findings reported in this paper.

5.5.5 Automated Crash Reproduction

Depending on the available information and complexity of the reported crash, reproducing the crash may be a complex and time consuming task for developers. Researchers have proposed several approaches to automated crash reproduction. The state of the art techniques are: STAR [81], EVOCRASH [204], and JCHARMING [171].

Each of the proposed approaches have certain advantages and limitations, which are to some extent reported in [204]. Upon further advances in this direction, automated crash reproduction may compensate for lack of crash reproducing steps in bug reports.

5.5.6 What Do User Contents Provide?

The results show that user contents have statistically significant impact on bug resolution times for ∼33% of the projects. User contents are provided through a link in the bug reports. However, their contents vary. In our manual analysis, we found out that the links may refer to long stack traces that the users preferred to provide separately from the main bug report. It is also possible for user contents to address fix suggestions or UI features. Future work may investigate the kinds of data provided through user contents and their frequencies. Such investigation helps analyse the impact of user contents more accurately.

5.6 Related Work

To understand what makes a good bug report, Zimmermann et el. [224] conducted a survey among developers and users of Apache, Eclipse, and Mozilla. They found out that across all three projects, crash reproducing steps, and stack traces, are most useful. At the same time these types of information are most difficult for users to provide. Their results show, to a large extent, lack of tool support causes this mismatch. For example, while stack traces are hidden in log files, experienced users of Eclipse know that Error logs exists. Therefore, experienced users can provide stack traces while for other users it is difficult to do so [224].

In addition, Zimmermann et al. [224] asked developers to rate 289 bug reports, that were selected randomly, from very poor to very good, using a five-point Likert scale [153]. They use the rated bug reports to train the CUEZILLA approach they propose. CUEZILLA measures the quality of bug reports, and recommends which elements should be added to improve the quality of bug reports.

This paper builds on the work by Zimmermann et al. [224] in that we interviewed and surveyed developers to understand their perceptions on the importance of different bug report elements. However, while Zimmermann et al. [224] surveyed the developers and users of Apache, Eclipse, and Mozilla, our approach to finding interview and survey participants were different. We first found participants from ERP, E-commerce, and automotive industries to execute the interviews. We used the insights from the interviews to construct a survey study where we contacted active developers from 85 different trending projects on Github. Furthermore, while CUEZILLA uses developers' ratings to measure the quality of bug reports, *IMaChecker* takes a different approach for analyzing the bug reports. *IMaChecker* statically parses the bug reports from 250 projects (developed in five different languages) to identify which elements are present in the bug reports, and using this information, *IMaChecker* applies statistical tests to identify the impact of the bug report elements on bug resolution times. Our findings with regards to the impact of bug report elements on bug resolution times are aligned with the findings reported by Zimmermann et al. [224] in that the results from interviews, surveys, and statistical tests show crash reproduction steps and stack traces are most useful for processing bug reports. Furthermore, despite the indicated importance, our results show that the majority of times, these elements are not included in bug reports.

Schroter et al. [197] conducted an empirical study with the Eclipse project to understand the extent to which stack traces are useful when debugging. Their findings show that the average lifetimes of bug reports which include stack traces are significantly lower than of other bugs. Furthermore, their findings show up to 60% of bug reports which included stack traces involved changes to one of the stack frames.

In this paper, we expand the findings reported by Schroter et al. [197] in that we study bug reports from 250 projects to assess the impact of several different bug report elements, including crash stack traces. Our results on the importance of crash stack traces for bug resolution times are aligned with the findings reported by Schroter et al. [197].

With regard to characterizing bug report quality, Hooimeijer and Weimer [123] provide a descriptive model based on a statistical analysis of 27000 publicly available bug reports for the Mozilla Firefox project. The proposed model predicts whether a reported bug is fixed within a given amount of time.

With regards to estimating the time it take to fix a bug report, [222] present a non-parametric approach based on using dissimilarity matrix and self-organizing neural networks. They used NASA's KC1 data set to evaluate their approach. The results indicated that their clustering approach performs well when applied on a family of

products such as software projects in product lines. However, the defect fix estimation performed poorly when applied on software projects from different environments. Moreover, Weiss et al. [211] propose an approach that automatically predicts the time it takes to fix a bug. Given a new reported issue, their technique finds similar older issues and uses their resolution time for prediction. They evaluated their approach using effort data from JBoss project. For bug reports, their technique is off by one hour.

In this paper, rather than providing a prediction model for estimating the time it takes to fix a bug, we use statistical tests to show how different bug report elements impact the time it takes to close bug reports. Furthermore, rather than looking into a single case study, we studied bug reports from 250 open source projects from Github.

5.7 Conclusions

Software projects often have open issue repositories. Bug reports that are submitted to issue repositories have varying contents. Therefore it is important to gain understanding about the significance of different elements in bug reports.

To understand the extent to which developers perceive various types of information important, we interviewed 35 developers. To asses the findings, we further surveyed 305 developers. The results show crash description, reproducing steps, and stack traces are of high importance in developers' perceptions.

To identify how often the important information elements are provided in bug reports, and what their impact is on bug resolution times, we developed *IMaChecker* to mine and analyse issues from Github repositories. Our statistical analysis, on issues from 250 projects on Github, confirms that crash reproducing steps, stack traces, fix suggestions and user contents have statistically significant impact on bug resolution times. However, on average, over ∼70% of the bug reports of a given repository lack these elements. Future work may investigate means to support users and developers for providing high quality bug reports.

Appendix A

Table 5.4 shows the corpus of 250 open source projects we selected from Github.

Table 5.4: This table shows the repositories we use in the evaluation.

Repository	Since	Stars	Language	Forks	Contributors
30-seconds/30-seconds-of-code	2017	43.1k	Javascript	4.7k	164
achael/eht-imaging	2016	4.6k	Python	414	9
activeadmin/activeadmin	2010	8.4k	Ruby	2.9k	569
adam-p/markdown-here	2012	37.3k	Javascript	6.3k	12
adobe/brackets	2011	29.7k	Javascript	6k	355
ageitgey/face_recognition	2017	23.7k	Python	6.2k	23
airbnb/lottie-android	2016	25.3k	Java	3.9k	71
androidannotations/ androidannotations	2010	10.7k	Java	2.4k	56
angular/angular.js	2010	59.5k	Javascript	28.9k	1595
ansible/ansible	2012	37.1k	Python	15.1k	4372
apache/incubator-dubbo	2012	25.9k	Java	17.2k	198
apache/incubator-echarts	2013	33.6k	Javascript	9.8k	71
apache/incubator-zipkin	2015	10.9k	Java	1.9k	78
atech/postal	2017	8.9k	Ruby	522	14
atom/atom	2011	48.5k	Javascript	11.4k	431
axios/axios	2014	58.2k	Javascript	4.5k	164
aymericdamien/TensorFlow-Examples	2015	30.9k	Python	11.7k	54
babel/babel	2012	32.8k	Javascript	3.4k	724
barryvdh/laravel-debugbar	2013	9.3k	PHP	905	95
barryvdh/laravel-ide-helper	2013	8.2k	PHP	782	107
bazelbuild/bazel	2015	11.9k	Java	1.9k	441
bcit-ci/CodeIgniter	2006	17.2k	PHP	7.6k	441
BetterErrors/better_errors	2012	6.5k	Ruby	430	75
binux/pyspider	2014	13k	Python	3.2k	51
bobthecow/psysh	2012	7.4	PHP	216	48
briannesbitt/Carbon	2012	12.4k	PHP	1k	197
bumptech/glide	2013	26k	Java	9k	96
CachetHQ/Cachet	2014	9.6k	PHP	1.1k	161
cakephp/cakephp	2005	7.8k	PHP	3.4k	523
capistrano/capistrano	2013	11k	Ruby	1.7k	215
carrierwaveuploader/ carrierwave	2008	8.3k	Ruby	1.4k	326
celery/celery	2009	12.3k	Python	3.2k	714
certbot/certbot	2012	25k	Python	2.5k	352
chartjs/Chart.js	2013	43k	Javascript	9.5k	298
chrisbanes/PhotoView	2012	15.2k	Java	3.5k	34
CocoaPods/CocoaPods	2011	11.6k	Ruby	2k	266
composer/composer	2011	19.6k	PHP	5.4k	729
daimajia/AndroidSwipe Layout	2014	11.1k	Java	2.6k	16
daimajia/AndroidView Animations	2014	10.5k	Java	2.2k	17
deeplearning4j/deep learning4j	2013	10.7k	Java	4.6k	250
deployphp/deployer	2013	6.7k	PHP	977	174
diaspora/diaspora	2010	12.2k	Ruby	2.9k	342
dingo/api	2014	8.3k	PHP	1.1k	96
docker/compose	2013	16k	Python	2.4k	299
doctrine/inflector	2009	7k	PHP	90	55
doctrine/instantiator	2014	6.8k	PHP	42	22
doctrine/lexer	2013	6.8k	PHP	29	16
Dogfalo/materialize	2014	35.6k	Javascript	4.7k	252
donnemartin/system-design-primer	2017	62.9k	Python	9.2k	65
egulias/EmailValidator	2013	6.7k	PHP	91	37
elastic/elasticsearch	2010	40.3k	Java	13.4k	1205
elastic/logstash	2009	10.2k	Ruby	2.7k	398
encode/django-rest-framework	2010	14k	Python	4.1k	851
EnterpriseQualityCoding /FizzBuzzEnterpriseEdition	2012	10.8k	Java	505	30
erusev/parsedown	2013	10.8k	PHP	881	39
eugenp/tutorials	2013	14k	Java	20.4k	500
explosion/spaCy	2014	13.2k	Python	2.2k	333
expressjs/express	2009	43.4k	Javascript	7.4k	220
facebook/create-react-app	2016	66.5k	Javascript	14.8k	672
facebook/fresco	2015	15.5k	Java	3.6k	152
facebook/react	2013	127k	Javascript	23.2k	1296
facebook/react-native	2015	76.2k	Javascript	17k	1947
facebook/stetho	2015	11k	Java	1k	49
facebookresearch/Detectron	2018	20.3k	Python	4.3k	27
faif/python-patterns	2012	20.4k	Python	4.4k	86
fastlane/fastlane	2014	25.4k	Ruby	3.8k	961
filp/whoops	2013	10k	PHP	523	99
fluent/fluentd	2011	7.8k	Ruby	913	169
FortAwesome/Font-Awesome	2018	59.5k	Javascript	10k	5

freeCodeCamp/devdocs	2013	20.5k	Ruby	1.3k	93
freeCodeCamp/freeCode Camp	2013	302k	Javascript	21.6k	3532
FriendsOfPHP/Goutte	2010	7.2k	PHP	871	66
FriendsOfPHP/PHP-CS-Fixer	2012	7.5k	PHP	203k	1k
gatsbyjs/gatsby	2015	33.9k	Javascript	4.8k	1954
getgrav/grav	2014	10.8k	PHP	1k	148
getredash/redash	2013	12.5k	Python	2k	247
getsentry/sentry	2008	20.7k	Python	2.3k	383
github/linguist	2011	6.7k	Ruby	2.4k	748
gollum/gollum	2010	9.9k	Ruby	1.4k	144
google-research/bert	2018	14.8k	Python	3.4k	26
google/ExoPlayer	2014	12.9k	Java	3.9k	135
google/gson	2008	15.5k	Java	3.1k	93
google/guava	2011	31.1k	Java	7k	185
google/python-fire	2017	14k	Python	818	28
GoogleChrome/puppeteer	2017	48.2k	Javascript	4.2k	208
greenrobot/greenDAO	2011	11.2k	Java	2.7k	6
gulpjs/gulp	2013	31.1k	Javascript	4.4k	216
guzzle/guzzle	2011	16.6k	PHP	1.9k	294
h5bp/html5-boilerplate	2010	42.6k	Javascript	10.1k	231
hakimel/reveal.js	2011	45.8k	Javascript	13.2k	245
hashicorp/vagrant	2010	18.4k	Ruby	3.7k	884
hdodenhof/CircleImageView	2014	11.7k	Java	2.6k	12
HelloZeroNet/ZeroNet	2015	13.7k	Python	1.7k	101
home-assistant/home-assistant	2013	23.5k	Python	6.8k	1441
Homebrew/brew	2009	17.6k	Ruby	3.9k	669
Homebrew/homebrew-cask	2012	15.2k	Ruby	7.2k	5214
huge-success/sanic	2016	12k	Python	1.1k	206
huginn/huginn	2013	21.3k	Ruby	2.3k	171
iluwatar/java-design-patterns	2014	46.8k	Java	15.1k	145
imathis/octopress	2009	9.5k	Ruby	2.9k	111
impress/impress.js	2011	34.7k	Javascript	6.8k	63
Intervention/image	2013	9.2k	PHP	1k	71
ipython/ipython	2008	13.5k	Python	3.8k	593
JakeWharton/butterknife	2013	23.7k	Java	4.5k	83
jakubroztocil/httpie	2012	41.1k	Python	2.6k	74
javan/whenever	2009	7.9k	Ruby	685	82
jekyll/jekyll	2008	37.7k	Ruby	8.2k	852
jfeinstein10/SlidingMenu	2012	11.1k	Java	5.3k	21
jordansissel/fpm	2011	9.1k	Ruby	915	234
josephmisiti/awesome-machine-learning	2014	39.8k	Python	9.7k	371
jquery/jquery	2006	51.4k	Javascript	18k	275
juliangarnier/anime	2016	30.7	Javascript	2.2k	27
kaminari/kaminari	2011	7.4k	Ruby	958	133
kennethreitz/requests	2011	38.6k	Python	6.9k	533
keon/algorithms	2016	14.9k	Python	2.7k	105
keras-team/keras	2015	41.1k	Python	15.3k	795
kilimchoi/engineering-blogs	2015	15.2k	Ruby	1.7k	303
Konloch/bytecode-viewer	2014	10.1k	Java	637	15
laravel/framework	2013	17.2k	PHP	6.4k	1944
lgvalle/Material-Animations	2015	12.7k	Java	2.5k	9
LMAX-Exchange/disruptor	2011	10.3k	Java	2.6k	31
localstack/localstack	2016	16.7k	Python	1.1k	157
lodash/lodash	2009	38.7k	Javascript	22.5k	280
loopj/android-async-http	2011	10.4k	Java	4.2k	75
Maatwebsite/Laravel-Excel	2013	6.9k	PHP	1.1k	95
magento/magento2	2011	7.3k	PHP	6.3k	1129
mame/quine-relay	2013	7.8k	Ruby	383	12
matomo-org/matomo	2007	11.1k	PHP	1.7k	224
matterport/Mask_RCNN	2017	11.9k	Python	5.1k	40
meteor/meteor	2011	41k	Javascript	5k	402
Microsoft/vscode	2015	73.9k	Javascript	10k	871
middleman/middleman	2009	6.4k	Ruby	694	182
mikepenz/MaterialDrawer	2014	10.3k	Java	2k	87
minimaxir/big-list-of-naughty-strings	2015	32.3k	Python	1.3k	56
mitmproxy/mitmproxy	2010	14.9k	Python	1.9k	253
mockery/mockery	2009	7.7k	PHP	356	139
moment/moment	2011	40.9k	Javascript	6.1k	492
monicahq/monica	2017	7.1k	PHP	838	158
mperham/sidekiq	2012	9.6k	Ruby	1.6k	397
mrdoob/three.js	2010	50.7k	Javascript	19k	1077
mui-org/material-ui	2014	46.1k	Javascript	9.9k	1229
mybatis/mybatis-3	2010	10.6k	Java	6.6k	121
NARKOZ/hacker-scripts	2015	34.9k	Javascript	5.9k	41
NationalSecurityAgency /ghidra	2019	15.3k	Java	1.8k	40

netty/netty	2008	18.8k	Java	8.4k	373
nextcloud/server	2010	7.4k	PHP	1.3k	601
nicolargo/glances	2011	13.3k	Python	906	92
nikic/PHP-Parser	2011	10.4k	PHP	614	82
nodejs/node	2009	60.3k	Javascript	13.4k	2444
nostra13/Android-Universal-Image-Loader	2011	16.4k	Java	6.3k	35
nvbn/thefuck	2015	43.7k	Python	2.1k	123
octobercms/october	2013	8.5k	PHP	1.9k	303
omniauth/omniauth	2010	6.8k	Ruby	870	143
openai/gym	2016	16.6k	Python	4.4k	176
orhanobut/logger	2015	11.1k	Java	1.7k	10
overtrue/wechat	2015	7.8k	PHP	1.9k	98
pallets/flask	2010	44k	Python	12.2k	507
pandas-dev/pandas	2009	19.4k	Python	7.7k	1479
parcel-bundler/parcel	2017	31.3k	Javascript	1.4k	204
phalcon/cphalcon	2012	9.6k	PHP	1.7k	226
phanan/koel	2015	10.2k	PHP	1.2k	45
PhilJay/MPAndroidChart	2014	27k	Java	7k	67
php-ai/php-ml	2016	6.8k	PHP	947	28
PHPMailer/PHPMailer	2008	13.1k	PHP	7.2k	168
plataformatec/devise	2009	19.9k	Ruby	4.6k	541
plataformatec/simple_form	2009	7.3k	Ruby	1.2k	219
prettier/prettier	2916	31.4k	Javascript	1.7k	413
pypa/pipenv	2017	16.8k	Python	1.2k	276
rails/rails	2004	43.1k	Ruby	17.3k	3818
ramsey/uuid	2012	8.7k	PHP	315	59
rapid7/metasploit-framework	2005	16.3k	Ruby	8.1k	628
react-native-community/lottie-react-native	2016	11.2k	Java	1k	53
ReactiveX/RxAndroid	2013	17.9k	Java	2.8k	59
ReactiveX/RxJava	2012	38.6k	Java	6.5k	240
reactphp/react	2012	6.8k	PHP	672	29
ReactTraining/react-router	2014	35.9k	Javascript	7.3k	548
realm/realm-java	2012	10.4k	Java	1.6k	80
reduxjs/redux	2015	48.1k	Javascript	12.3k	673
resque/resque	2009	8.5k	Ruby	1.5k	207
resume/resume.github.com	2011	40.3k	Javascript	1k	48
roots/sage	2011	10k	PHP	2.8k	193
rubocop-hq/rubocop	2012	9.9k	Ruby	2k	596
ruby-grape/grape	2010	8.8k	Ruby	1k	319
ryanb/cancan	2009	6.3k	Ruby	839	54
scikit-learn/scikit-learn	2010	35k	Python	16.9k	1304
scrapy/scrapy	2008	32.8k	Python	7.6k	313
sebastianbergmann/phpunit	2006	13.8k	PHP	1.7k	358
Seldaek/monolog	2011	14.6k	PHP	1.5k	324
SeleniumHQ/selenium	2004	14.3k	Java	4.8k	429
Semantic-Org/Semantic-UI	2013	45.2k	Javascript	4.8k	190
serbanghita/Mobile-Detect	2012	8.6k	PHP	2.3k	83
serverless/serverless	2015	29.9k	Javascript	3k	571
sferik/rails_admin	2010	7k	Ruby	2k	357
Shopify/liquid	2008	7.1k	Ruby	931	121
signalapp/Signal-Android	2011	11.4k	Java	2.9k	183
sinatra/sinatra	2007	10.6k	Ruby	1.9k	361
skylot/jadx	2013	18.5k	Java	2k	31
slimphp/Slim	2010	9.8k	PHP	1.8k	202
socketio/socket.io	2004	46k	Javascript	8.5k	154
spree/spree	2008	9.7k	Ruby	4.2k	252
spring-projects/spring-boot	2012	36.8k	Java	24.1k	571
spring-projects/spring-framework	2008	29k	Java	19k	364
sqlmapproject/sqlmap	2008	14.1k	Python	3k	79
square/okhttp	2012	31.8k	Java	7k	182
square/picasso	2013	16.7k	Java	3.9	91
square/retrofit	2010	32k	Java	5.9k	125
StevenBlack/hosts	2012	12k	Python	1.1k	61
storybooks/storybook	2015	36.7k	Javascript	2.9k	677
stympy/faker	2007	7.7k	Ruby	1.9k	585
swiftmailer/swiftmailer	2007	7.8k	PHP	738	133
symfony/symfony	2010	20.6k	PHP	6.8k	1866
teamcapybara/capybara	2009	8.7k	Ruby	1.2k	259
Tencent/tinker	2016	13.6k	Java	2.7k	22
tensorflow/magenta	2016	13.1k	Python	2.5k	97
tensorflow/models	2016	52.6k	Python	31.7k	497
the-control-group/voyager	2016	8k	PHP	1.9k	288
TheAlgorithms/Java	2016	13.5k	Java	5k	137
TheAlgorithms/Python	2016	38.4k	Python	10.9k	218
thedaviddias/Front-End-Checklist	2017	34.2k	Javascript	3.2k	82
thephpleague/flysystem	2013	9.3k	PHP	512	171
thepracticaldev/dev.to	2018	9.1k	Ruby	1k	187

thoughtbot/bourbon	2011	8.8k	Ruby	918	101
thoughtbot/factory_bot	2008	6.4k	Ruby	1.7k	187
thoughtbot/guides	2012	8k	Ruby	1.2k	98
thoughtbot/paperclip	2008	9k	Ruby	2k	371
tmuxinator/tmuxinator	2010	8.9k	Ruby	549	109
toddmotto/public-apis	2016	56.8k	Python	5.7k	475
tootsuite/mastodon	2016	17.6k	Ruby	3.1k	558
tornadoweb/tornado	2009	17.7k	Python	4.9k	304
trailofbits/algo	2016	13.1k	Python	1.1k	119
trekhleb/Javascript-algorithms	2018	47.9k	Javascript	6.7k	89
TryGhost/Ghost	2013	29.6k	Javascript	6.3k	314
twbs/bootstrap	2011	133k	Javascript	64.9k	1074
twbs/bootstrap-sass	2011	12.7k	Ruby	3.5k	95
tymondesigns/jwt-auth	2014	7.7k	PHP	968	65
typicode/json-server	2013	39.6k	Javascript	3.5k	61
udacity/fullstack-nanodegree-vm	2015	263	Python	11.3k	7
Valloric/YouCompleteMe	2012	19k	Python	2.1k	136
varvet/pundit	2012	6.4k	Ruby	489	92
vinta/awesome-python	2014	67.2k	Python	12.7k	306
vlucas/phpdotenv	2013	9.2k	PHP	439	47
vuejs/vue	2016	136k	Javascript	19.3k	274
walkor/Workerman	2013	7.4k	PHP	1.9k	49
webpack/webpack	2012	48.3k	Javascript	6k	516
wix/react-native-navigation	2016	10.2k	Java	2.2k	280
yarnpkg/yarn	2016	35.5k	Javascript	2.1k	496
yiisoft/yii2	2011	12.8k	PHP	6.7k	961
ytdl-org/youtube-dl	2008	50.3k	Python	8.5k	671
zeit/next.js	2016	36.7k	Javascript	4.2k	700
zxing/zxing	2013	22.4k	Java	8.1k	94

Appendix B

Table 5.5 presents the results of mining 250 issue repositories in detail.

Table 5.5: This table shows the frequencies of various elements in bug reports for different projects. **O/C** indicates the ratio between open issues and closed one. **bugA** indicates that the issues is originally labeled as a bug, whereas **bugB** indicates that *IMaChecker* detected the issue as a bug. **ST** indicates Stack Traces, **RS** indicates Reproducing Steps, **FS** indicates Fix Suggestions, **UC** shows User Content, and **C** shows Code.

Repository	#O/C	#bugA	#bugB	#ST	#RS	#FS	#UC	#C
30-seconds/30-seconds-of-code	9/184	23	49	0	17	32	11	1
achael/eht-imaging	1	0	2	2	0	0	0	0
activeadmin/activeadmin	34/541	203	527	317	130	101	17	4
adam-p/markdown-here	227/284	86	21	0	6	15	14	1
adobe/brackets	1169/3474	160	1148	4	923	238	253	2
ageitgey/face_recognition	302/431	0	175	114	41	26	44	1
airbnb/lottie-android	33/853	3	158	75	67	21	70	0
android annotations/android annotations	43/1564	0	253	150	67	52	3	3

angular/ angular.js	129/2849	1622	1214	14	944	727	37	3
ansible/ansible	280/1349	13982	15790	3367	14834	612	97	83
apache/ incubator -dubbo	331/2032	99	394	90	294	27	140	3
apache/ incubator -echarts	3209/6916	566	1442	18	1397	28	1309	5
apache/incubator -zipkin	13/41	88	95	45	23	29	49	0
atech/postal	93/161	20	52	27	2	24	49	2
atom/atom	557/14128	2937	5693	816	5343	672	600	5
axios/axios	515/1168	86	138	50	37	56	99	9
aymericdamien/ TensorFlow- Examples	129/53	0	21	20	0	2	3	0
babel/babel	625/5711	795	852	142	477	492	146	18
barryvdh/ laravel-debugbar	107/120	0	28	7	6	15	27	0
barryvdh/ laravel-ide-helper	294/257	0	28	6	4	18	24	3
bazelbuild/ bazel	875/2251	1931	1724	217	1426	177	27	18
bcit-ci/ CodeIgniter	45/2944	161	132	4	33	97	17	2
BetterErrors/ better_errors	42/209	16	46	37	1	8	3	0
binux/pyspider	219/529	6	166	110	50	17	11	1
bobthecow/psysh	55/282	39	16	4	6	7	11	1
briannesbitt/ Carbon	7/698	15	45	5	29	12	5	3
bumptech/glide	137/3205	395	538	414	71	87	129	5
CachetHQ/ Cachet	127/1859	432	266	129	100	50	82	1
cakephp/cake php	65/1704	30	379	68	88	231	31	10
capistrano/ capistrano	18/509	77	220	141	56	42	0	2
carrierwave uploader/carrier wave	169/1489	30	236	165	26	51	4	1
celery/celery	383/3441	376	1476	884	751	117	32	11
certbot/certbot	352/1629	470	837	606	69	184	11	4
chartjs/ Chart.js	424/4137	1003	684	6	497	382	353	11
chrisbanes/ PhotoView	116/451	39	59	44	7	10	10	0
CocoaPods/ CocoaPods	178/6943	0	1778	1168	482	191	63	15
composer/ composer	87/1280	663	577	35	163	395	45	14
daimajia/ AndroidSwipe Layout	30/13	9	27	15	5	7	5	1
daimajia/Android ViewAnimations	14/23	1	9	6	1	2	0	0
deeplearning4j/ deeplearning4j	391/1780	967	533	306	138	108	122	6
deployphp/ deployer	20/93	171	310	18	274	44	8	5
diaspora/ diaspora	418/4141	1761	564	264	176	149	33	2
dingo/api	67/359	52	133	52	57	28	6	2
docker/ compose	505/3891	511	970	461	488	109	24	17
doctrine/inflector	4/23	0	0	0	0	0	0	0
doctrine/instantiator	1/9	3	1	0	1	0	0	0
doctrine/lexer	Inf	0	0	0	0	0	0	0
Dogfalo/ materialize	481/4486	279	533	22	390	222	176	7
donnemartin/ system-design -primer	29/14	9	2	1	0	1	2	0
egulias/Email Validator	21/83	24	2	0	1	1	1	0
elastic/ elasticsearch	1736/18139	3510	5540	1719	3898	832	165	47

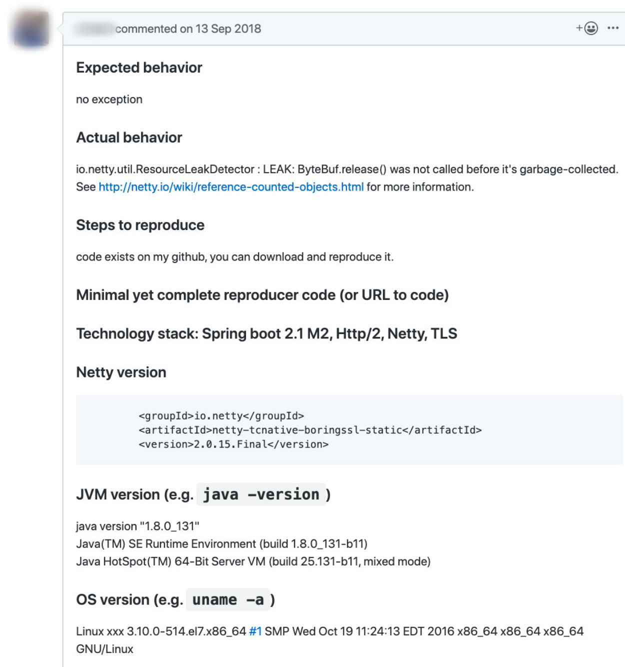
elastic/log stash	1366/3745	965	1071	506	457	200	41	7
encode/django -rest-framework	143/3023	311	812	158	608	120	35	3
EnterpriseQuality Coding/FizzBuzz EnterpriseEdition	134/27	1	15	2	0	13	0	0
erusev/parsedown	2/29	89	23	0	7	16	6	4
eugenp/tutorials	2/65	0	27	15	1	12	4	2
explosion/spaCy	70/1363	411	896	444	452	125	65	14
expressjs/ express	111/2956	90	178	48	58	73	29	4
facebook/ create-react-app	317/4174	400	1453	177	1243	181	338	12
facebook/fresco	79/1897	282	351	267	75	26	61	3
facebook/react	503/6964	369	1435	55	1212	204	217	10
facebook/react-native	219/7931	1319	4847	285	4329	436	1490	46
facebook/stetho	1/9	39	62	54	8	3	11	1
facebook research/Detectron	213/571	6	256	139	130	24	36	0
faif/python-patterns	9/47	1	3	1	0	2	1	0
fastlane/ fastlane	171/9137	522	2085	1754	69	332	140	24
filp/whoops	42/251	0	18	3	0	15	4	0
fluent/fluentd	197/960	106	290	232	52	36	16	3
FortAwesome/ Font-Awesome	4769/9638	385	456	6	143	312	1079	4
freeCodeCamp/ devdocs	77/647	102	51	14	21	17	24	0
freeCodeCamp/ freeCodeCamp	75/4444	261	7730	26	7465	366	944	9
FriendsOfPHP/ Goutte	105/131	0	9	4	3	2	3	0
FriendsOfPHP/ PHP-CS-Fixer	234/1373	247	77	9	23	48	8	4
gatsbyjs/gatsby	419/6047	534	2005	148	1611	564	442	27
getgrav/grav	103/556	219	121	24	39	59	75	5
getredash/redash	43/148	220	499	120	407	36	102	2
getsentry/sentry	68/253	73	489	291	152	72	184	0
github/linguist	5/417	16	62	17	7	38	30	0
gollum/gollum	14/449	212	114	66	24	25	4	0
google/Exo Player	294/4849	841	1334	756	524	171	112	3
google/gson	8/19	2	345	92	273	30	6	1
google/guava	371/1104	0	249	27	90	136	7	0
google/ python-fire	23/31	13	14	8	1	6	1	1
google-research/bert	11/8	0	71	42	13	21	23	0
GoogleChrome/ puppeteer	376/2539	186	1271	227	1115	83	226	26
greenrobot/ greenDAO	31/181	35	67	54	6	12	15	1
gulpjs/gulp	17/1707	27	116	47	20	52	22	3
guzzle/guzzle	127/597	24	194	30	138	40	17	3
h5bp/html5-boilerplate	1/1142	38	78	1	12	65	0	0
hakimel/ reveal.js	370/1269	49	80	3	35	42	19	1
hashicorp/ vagrant	199/3767	1600	2657	869	1738	256	54	14
hdodenhof/Circle ImageView	9/275	3	35	29	2	5	10	0
HelloZeroNet/ ZeroNet	535/1064	98	231	82	118	41	56	0
home-assistant /home-assistant	1007/8670	143	4607	2641	2840	158	545	19
Homebrew/brew	3/707	86	647	239	406	111	19	13
Homebrew/ homebrew-cask	1/222	18	1090	956	31	138	20	3
huge-success/ sanic	21/223	42	119	91	14	21	11	1
huginn/huginn	288/1207	20	223	160	9	65	20	2
iluwatar/java-design-patterns	181/263	40	9	2	0	7	0	0

imathis/ octopress	179/808	70	140	101	16	30	0	1
impress/ impress.js	46/379	5	28	0	16	12	0	0
Intervention/ image	229/500	8	29	6	4	19	19	1
ipython/ipython	144/611	930	1118	679	266	240	73	8
JakeWharton/ butterknife	39/484	9	103	65	12	30	32	4
jakubroztocil/ httpie	13/41	81	79	61	8	13	9	0
javan/whenever	55/463	25	87	69	2	16	0	0
jekyll/jekyll	67/3872	23	833	318	421	136	25	2
jfeinstein10/ SlidingMenu	263/381	6	72	43	13	21	1	0
jordansissel/fpm	14/13	64	188	146	24	34	2	0
josephmisiti/ awesome-machine-learning	1/57	0	3	0	1	2	0	0
jquery/jquery	13/358	103	145	5	84	59	38	1
juliangarnier/ anime	68/377	23	34	1	25	8	15	2
kaminari/kaminari	5/146	57	66	42	16	11	1	1
kennethreitz/ requests	25/391	137	655	512	107	94	27	6
keon/algorithms	14/29	7	5	1	0	4	3	0
keras-team/ keras	2323/6809	112	2479	1044	1324	389	212	21
kilimchoi/ engineering-blogs	7/59	0	0	0	0	0	0	0
Konloch/ bytecode-viewer	29/154	35	37	27	3	8	3	0
laravel/ framework	29/4011	346	3704	197	3255	431	176	49
lgvalle/Material-Animations	1/2	0	2	1	0	1	0	0
LMAX-Exchange/ disruptor	6/187	2	33	10	16	8	3	2
localstack/ localstack	410/489	199	256	210	38	26	11	4
lodash/lodash	0	227	154	12	49	94	40	2
loopj/android-async-http	243/799	0	134	107	2	28	2	0
Maatwebsite/ Laravel-Excel	27/1832	18	418	18	378	42	104	8
magento/ magento2	557/6754	3604	9383	664	8826	445	1787	15
mame/quine-relay	1/8	0	3	0	0	3	0	0
matomo-org/matomo	1717/8490	4102	920	85	439	416	301	3
matterport/ Mask_RCNN	857/483	0	203	143	4	61	99	5
meteor/meteor	290/7729	553	994	250	506	300	90	16
Microsoft/ vscode	2743/31334	17233	28328	503	25881	3228	10134	119
middleman/ middleman	123/1426	56	366	226	130	41	1	2
mikepenz/ MaterialDrawer	8/2269	120	180	100	40	45	89	4
minimaxir/big-list -of-naughty-strings	43/24	0	7	0	1	6	1	0
mitmproxy/ mitmproxy	53/425	368	1002	456	777	42	51	2
mockery/mockery	52/437	62	36	8	15	15	2	1
moment/moment	47/628	252	550	15	478	73	47	8
monicahq/ monica	332/717	199	65	15	22	29	103	0
mperham/ sidekiq	12/3073	0	575	414	69	128	31	13
mrdoob/ three.js	629/8187	900	571	28	172	378	322	8
mui-org/material-ui	281/8286	1286	3057	74	2716	868	779	21
mybatis/ mybatis-3	95/763	113	259	73	190	21	16	1

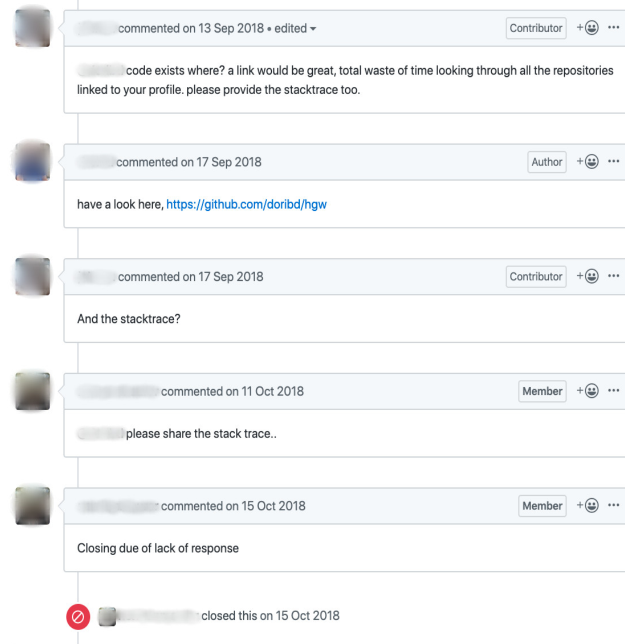
NARKOZ/hacker-scripts	9/8	0	2	0	1	1	0	0
NationalSecurity Agency/ghidra	241/276	258	176	6	163	14	109	1
netty/netty	388/4203	231	1332	565	776	112	46	1
nextcloud/server	1961/5781	3389	4114	546	3891	206	828	26
nicolargo/glances	107/955	345	284	258	22	23	30	3
nikic/PHP-Parser	41/343	0	33	8	7	18	2	1
nodejs/node	337/4612	645	1727	799	599	419	192	40
nostra13/Android-Universal-Image-Loader	273/472	89	181	145	10	31	2	0
nvbn/thefuck	37/80	3	156	76	60	48	11	0
octobercms/october	163/1164	717	779	23	696	85	179	13
omniauth/omniauth	29/263	15	87	57	19	17	2	0
openai/gym	175/689	1	191	136	21	43	22	1
orhanobut/logger	44/129	13	11	6	0	5	7	1
overtrue/wechat	22/719	8	17	17	0	0	72	3
pallets/flask	1/52	28	275	204	37	53	13	4
pandas-dev/pandas	963/4034	4242	1798	1144	354	377	138	16
parcel-bundler/parcel	321/715	779	598	126	387	349	151	9
phalcon/cphalcon	48/2011	630	351	82	184	100	19	5
phanan/koel	8/79	26	94	9	67	23	11	1
PhilJay/MP AndroidChart	201/389	82	314	174	40	112	342	2
php-ai/php-ml	26/35	0	10	4	0	6	7	0
PHPMailer/PHPMailer	29/1306	8	201	13	140	50	22	1
plataformatec/devise	5/764	147	586	416	87	114	11	5
plataformatec/simple_form	17/1099	61	76	39	15	25	8	0
prettier/prettier	583/2822	1142	220	41	126	60	96	41
pypa/pipenv	281/2355	332	877	733	118	80	46	9
rails/rails	371/12260	0	4799	1535	3541	361	86	30
ramsey/uuid	27/89	15	11	0	2	9	2	0
rapid7/metasploit-framework	637/2466	1032	1265	531	892	65	91	1
react-native-community/lottie-react-native	29/151	23	63	10	46	8	29	6
ReactiveX/RxAndroid	0	0	35	22	3	12	3	0
ReactiveX/RxJava	14/1333	237	347	164	95	102	13	8
reactphp/react	0	14	10	1	1	8	0	0
ReactTraining/react-router	7/988	141	871	25	731	138	90	10
realm/realm-java	425/3378	589	1161	568	694	102	38	14
reduxjs/redux	26/1615	16	209	8	113	96	33	1
resque/resque	34/769	113	104	79	12	19	2	0
resume/resume.github.com	29/54	11	6	0	4	2	0	0
roots/sage	17/1124	40	123	7	93	29	4	5
rubocop-hq/rubocop	225/2999	615	1312	341	1000	170	13	10
ruby-grape/grape	59/244	313	121	87	15	25	2	0
ryanb/cancan	68/215	15	84	50	3	33	0	0
scikit-learn/scikit-learn	1253/4945	930	1748	628	1111	248	61	12
scrapy/scrapy	59/149	141	384	282	42	79	17	4
sebastianberg mann/phpunit	69/2329	170	187	55	71	68	25	5
Seldaek/monolog	7/85	35	37	12	5	21	2	1
SeleniumHQ/selenium	365/5286	0	2941	251	2701	162	157	16

Semantic-Org/ Semantic-UI	781/5176	1190	487	22	322	149	180	5
serbanghita/ Mobile-Detect	36/109	78	39	0	0	39	5	1
serverless/ serverless	497/3000	793	363	209	72	93	45	11
sferik/rails _admin	340/1787	151	258	192	24	55	14	0
Shopify/liquid	27/106	46	36	16	10	10	4	2
signalapp/ Signal-Android	282/6443	104	2417	224	2064	597	345	2
sinatra/sinatra	65/638	86	98	64	18	21	0	0
skylot/jadx	39/188	31	29	21	4	5	33	0
slimphp/Slim	2/713	54	95	28	15	54	6	1
socketio/socket .io	377/2333	108	343	20	265	219	26	2
spree/spree	33/1186	0	838	394	404	152	31	2
spring-projects/spring -boot	381/13093	1375	2164	802	1187	298	110	30
spring-projects/spring -framework	731/17331	4006	1422	323	505	656	13	2
sqlmapproject/ sqlmap	4/281	828	1611	1451	226	80	26	1
square/okhttp	85/1228	508	611	492	107	60	33	5
square/picasso	27/214	8	207	161	25	30	9	0
square/retrofit	29/1051	19	295	230	31	45	19	5
StevenBlack/ hosts	28/307	17	42	23	3	16	19	1
storybooks/ storybook	111/1010	706	1106	105	992	87	405	15
stympy/faker	47/334	35	49	30	13	9	9	0
swiftmailer/ swiftmailer	151/223	0	87	29	37	23	3	3
symfony/ symfony	635/11437	2787	1747	109	1225	560	268	35
teamcapybara/ capybara	1/1302	78	304	156	146	44	7	0
Tencent/tinker	69/440	63	198	197	0	3	34	0
tensorflow/ magenta	173/381	0	132	109	5	25	28	0
tensorflow/ models	1269/2959	112	1904	913	1226	141	207	10
the-control-group/voyager	66/773	470	1289	6	1239	68	350	7
TheAlgorithms/ Java	1/10	2	6	0	0	6	1	1
TheAlgorithms/ Python	35/86	3	5	2	0	3	3	0
thedaviddias/ Front-End-Checklist	3/88	14	7	0	1	6	7	0
thephpleague/ flysystem	9/520	5	52	15	18	21	2	4
thepracticaldev/ dev.to	11/35	332	317	9	274	39	291	3
thoughtbot/ bourbon	1/275	18	45	8	24	15	0	0
thoughtbot/ factory_bot	18/767	25	129	88	15	30	3	1
thoughtbot/guides	1/15	0	3	0	0	3	1	0
thoughtbot/ paperclip	15/1801	138	243	170	24	58	3	1
tmuxinator/ tmuxinator	80/321	41	50	28	13	10	1	0
toddmotto/public-apis	8/95	0	10	0	1	9	2	0
tootsuite/ mastodon	1175/3353	779	741	169	270	325	412	8
tornadoweb/ tornado	142/1301	0	313	251	39	51	9	2
trailofbits/algo	4/109	77	478	46	454	28	22	1
trekhleb/ Javascript-algorithms	33/68	13	9	0	4	5	5	0
TryGhost/Ghost	81/5305	1424	1659	90	1459	215	205	5
twbs/ bootstrap	329/18145	200	1185	18	473	708	601	13

twbs/bootstrap-sass	3/811	39	71	36	15	23	4	1
tymondesigns/jwt-auth	397/892	6	158	21	101	42	24	2
typicode/json-server	199/183	3	42	21	7	15	15	5
udacity/fullstack-nanodegree-vm	3/2	0	4	1	3	3	3	0
Valloric/You CompleteMe	42/2609	5	927	376	546	181	62	1
varvet/pundit	8/329	4	48	29	1	18	1	1
vinta/awesome-python	71/69	0	5	0	1	4	0	1
vlucas/phpdotenv	0	0	11	1	3	7	0	0
vuejs/vue	67/2606	330	3150	47	2991	176	146	26
walkor/Workerman	8/255	10	10	6	0	4	10	1
webpack/webpack	442/5893	811	2404	176	2222	185	195	21
wix/react-native-navigation	69/1738	119	1549	77	1461	75	459	18
yarnpkg/yarn	347/674	571	2627	309	2447	80	108	31
yiisoft/yii2	14/249	1638	2214	189	1763	376	87	28
ytdl-org/youtube-dl	2352/14395	474	9337	6980	127	4517	97	10
zeit/next.js	205/3877	208	1336	254	1064	324	319	16
zxing/zxing	2/223	40	123	48	53	29	70	1



(a) Snapshot of the bug report [1].



(b) Snapshot of the responses to the bug report [1].

Figure 5.1: An snapshot of a bug report [1] which is missing a crash stack trace, as well as the responses to it.

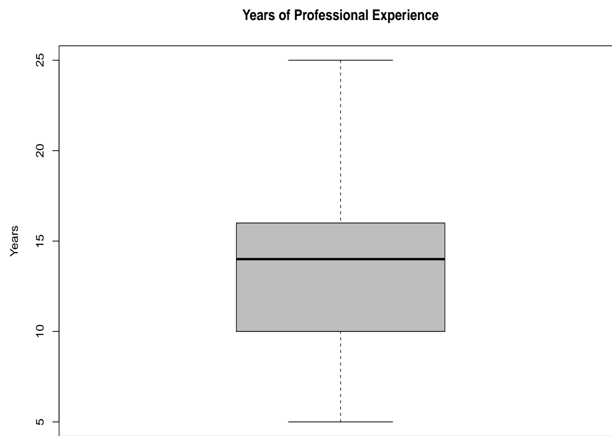


Figure 5.2: The years of professional experience of the interview participants.

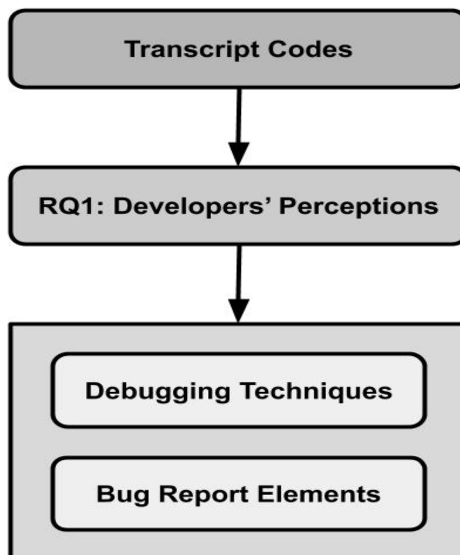


Figure 5.3: The identified themes after analysing the interview transcripts.

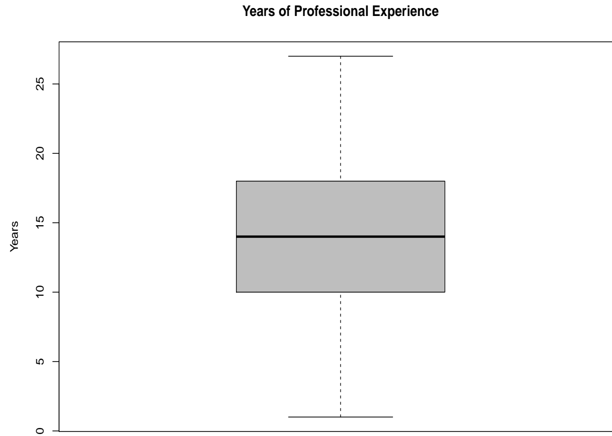


Figure 5.4: The figure presents the years of professional experience of the survey participants.

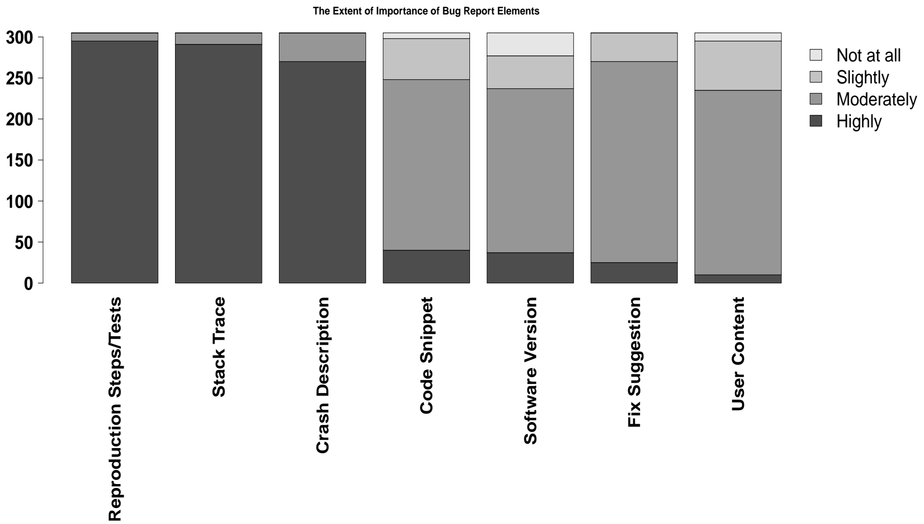


Figure 5.7: Developers' perception on the importance of various data for bug resolution time.

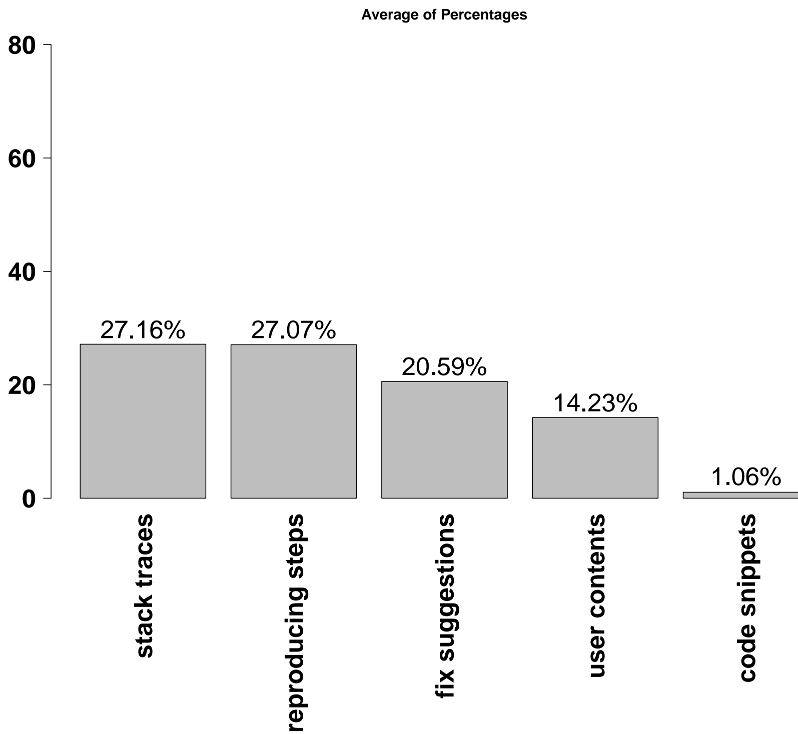


Figure 5.8: Average percentages of various elements of bug reports.

6 The Use of Contracts in Open Source Software

Design by Contract (DBC) is a software development approach in which contracts are formally specified between client and supplier components. Specified contracts can be used in program verification, automated testing, and API documentation, thereby, helping improve software quality. Assertions and other built-in features of programming languages, as a lightweight form of contracts, have been investigated in related work. However, the use of DBC in popular languages has been underexplored. In this study, we present results of an empirical evaluation on the use of contracts in 124 open source projects, written in Java, C++, and Python. Our findings show that the average use of different types of contracts differ depending on the program language. In addition, we derived the following use cases of contracts: checking null conditions, as well as checking equality and semantics of objects, data collections, strings, and numbers. Furthermore, the results of regression analysis shows there is a negative relation between the number of contracts and frequency of defect occurrence in a method. These results are statistically significant for all Java, C++, and Python projects.

6.1 Introduction

Delivering a reliable software product is a pressing concern in software development. Software reliability is especially important when software components are designed to

be reused in various applications. In order to support the goal of developing reliable software, Meyer [162] [163] presents pragmatic guidelines based on the theory of *Design By Contract (DBC)*.

The underlying idea in DBC is that software components are expected to collaborate based on mutual expectations and benefits, which are formally specified in software programs. In a program, if a client routine calls a supplier routine, the client must meet certain requirements on entry, which are typically referred to as *preconditions*. At the exit, the supplier routine may guarantee certain properties, named *postconditions*. *class invariants* are certain conditions which must be always guaranteed. Thus, program contracts are used for various purposes such as static and runtime verification, API documentation, and automated software testing [98].

Recently, Casalnuovo et al. [78] explored the connection between the use of assertions and occurrence of defects in C and C++ projects from Github. They report that the use of assertions does have small (yet significant) effect on reducing the defect density. In addition, they report that assertions tend to be added to methods by developers who have a larger ownership of those methods. Kochhar and Lo [144] performed a partial replication of the study by Casalnuovo et al. [78] on Java programs from Github. Their results confirm the findings reported by Casalnuovo et al. [78]. Furthermore, Estler et al. [98] study the use of contracts in 21 projects developed in Eiffel, C#, and Java. Their findings show that contracts are quite stable and may change infrequently. In addition, they report that there is no strong preference for different types of contracts.

In this study, we intend to broaden the scope of the studies which were previously performed with regards to the use of contracts. Therefore, we aim to investigate how often different types of contracts are used and for what purposes. In addition, similar to the studies by Casalnuovo et al. [78] and Kochhar and Lo [144] which analyze the relation between use of assertions and occurrence of defects, we aim to analyze the relation between the use of contracts and occurrence of defects. Therefore, we devise the following research questions:

- **RQ_1:** How often are different types of contracts used?
- **RQ_2:** For which use cases do developers use contracts?
- **RQ_3:** Does the use of contracts relate to occurrence of defects?

To answer the above research questions, we study three popular programming languages in Github [16] [17]: Java, Python and C++, which support *design by contract (DBC)*. Java and C++ are popular instances of strongly typed object-oriented and semi object-oriented programming languages, respectively. Python is a popular

instance of a high-level interpreted programming language. For each language we selected popular libraries that support DBC, namely: JML [31, 149], Valid4j [37], Cofoja [23], Boost.contract [20], Icontract [27, 146], and Pycontracts [34]. We selected projects that use these libraries from Github, thereby, we created a corpus that contains 124 projects.

For each library we developed a static source code parser which can detect and record the use of contracts. In addition, for each language, we developed a Git commit log parser, which can detect and record methods that are changed in commits. Moreover, we manually analyzed a subset of 1505 (out of 18494) automatically identified contracts that we randomly selected from each project.

Our results indicate that in Java projects, on average, 56% of the contracts are preconditions while 31% of them are postconditions, and 13% are class invariants. These results contradict the results reported by Estler et al. [98] showing that there is indeed strong preference for using preconditions, compared to postconditions and invariants. On the other hand, in C++ projects, on average, 46% of the contacts are postconditions, %29 are preconditions, and 19% are class invariants. For Python projects, on average, 77% of the contracts are preconditions, 16% of the contracts are postconditions, and 7% of the contracts are invariants. In addition, we derived five categories of use cases for contracts: contracts for objects, contracts for data collections, contracts for numbers, contracts for strings, and contracts for null conditions. These categories are further divided into: equality checks on objects, boundary checks and equality checks for numbers, semantic checks and equality checks for strings, size checks and semantic checks for data collections.

Regarding the analysis on the relation between use of contracts and occurrence of defects, we parsed the Git commit logs and filtered the fixing commits using the same heuristics used by Kochhar and Lo [144]. We used Poisson regression analysis [198] on the parsed commits. The results show negative relation between the use of contracts and occurrence of defects. For all Java, C++, and Python libraries, the results are statistically significant.

The contributions in this paper are the following:[1]

1. a set of six open source static parsers written in Python, for six different libraries that support DBC in Java, C++, and Python programs,
2. a set of three open source Git commit log parsers written in Python, for Java, C++, and Python projects,

[1] The corpus of projects, extracted contracts, extracted Git commit logs, the results of Git commit analysis and R scripts are provided through the following DOI link: 10.5281/zenodo.3610696
Upon acceptance of the paper, all contributions will be made publicly available.

3. the collection of 124 Java, C++, and Python projects in which DBC is used,
4. an extensive data set that contains the results of automated contract detection on 124 projects, and automated change detection from Git logs,
5. the results of the manual analysis on 1505 contracts which were automatically detected, and
6. all R scripts used to analyze the results.

The remainder of the chapter is organized as follows: Section 6.2 provides related work. Section 6.3 presents the research methodology. Section 6.4 presents the results. Section 6.5 and 6.6 provide discussion and threats to validity, respectively. Section 6.7 concludes the paper.

6.2 Related Work

In this section, we present the related work about the use of assertions and their effect on defect occurrence, in addition to the related work about empirical studies using projects which are hosted on Github.

6.2.1 Assertion Use and Impact on Quality

Prior to the analysis Casalnuovo et al. [78] performed on the use of assertions in Github projects, Kudrjavets et al. [147] performed an empirical case study on two software components from Microsoft to investigate the relation between software assertions and software faults. According to their observations, with an increase in the assertion density in a file, there is a statistically significant decrease in the fault density.

Later, Casalnuovo et al. [78] studied the use of assertions in Github projects which were developed in C and C++. They found out that assertions are widely used in popular C and C++ projects. They further explored the connection between the use of assertions and occurrence of defects in these projects. Therefore, they report that the use of assertions does have small, but significant, effect on reducing the frequency of defect occurrence. In addition, they report that assertions tend to be added to methods by developers who have a larger contribution to those methods.

Recently, Kochhar and Lo [144] performed a partial replication of the study by Casalnuovo et al. [78] on 185 Java projects from Github. Their results confirm the findings

reported by Casalnuovo et al. [78]. Additionally, they performed manual analysis and identified eight categories of assertions that developers used in the Java projects. The identified categories are: null condition check, process state check, initialization check, resource check, resource lock check, minimum and maximum value constraint check, collection data and length check, and implausible condition check.

In order to support development of reliable software, Meyer [162] [163] provided pragmatic techniques which are based on the theory of Design By Contract (DBC). Contracts are executable form of formal specifications, which are typically expressed as method preconditions, method postconditions, and class invariants. Estler et al. [98] performed an empirical study on 21 Eiffel, C# and Java projects to investigate which types of contracts are used more often, and how contracts are evolved over time. Estler et al. [98] report that they did not observe strong preference for a certain type of contract. However, when preconditions are used, they typically include more predicates than when postconditions are used. In addition, they observe that the use of contracts tends to be quite stable over time.

Moreover, Dietrich et al. [94] report from an empirical study on 200 Java programs and highlight that while the adoption of contracts has been slow in reality, the adoption of lightweight contracts through utilizing built-in features of programming languages and runtime checking has progressed. Thus a wide range of techniques and constructs are used to represent contracts. Often the same program uses different techniques at the same time. Therefore, Dietrich et al. [94] catalogue 25 techniques and tools for lightweight contract checking, using built-in features of the language such as assertions and exceptions.

6.2.2 Empirical Studies on Github Projects

Jiang et al. [133] explore why and how developers fork what from whom in GitHub. Therefore they collect a dataset containing 236,344 developers and 1,841,324 forks. Their observations indicate developers fork repositories to submit pull requests, fix bugs, add new features and keep copies. Developers find repositories to fork from various sources: search engines, external sites (e.g., Twitter, Reddit), social relationships, etc. More than 42 % of developers that they surveyed agree that an automated recommendation tool is useful to help them pick repositories to fork, while more than 44.4 % of developers do not value a recommendation tool. Moreover, their findings indicate that a repository written in a developer's preferred programming language is more likely to be forked.

Kochhar et al. [141] [142] explore 50,000 projects and investigate the correlation

between the presence of test cases and various project development characteristics, including the lines of code and the size of development teams. According to their findings, projects having test cases are bigger in size than projects without test cases. However, as projects get larger the number of tests per line of code decreases. Moreover, they report that projects having bigger team size have higher number of test cases whereas the number of test cases per developer decreases with an increase in the size of the development team.

Kochhar et al. [145] perform a large scale empirical study where they gather a large dataset consisting of popular projects from Github (628 projects, 85 million SLOC, 134 thousand authors, 3 million commits, in 17 languages) to understand the impact of using multiple languages on software quality. They build multiple regression models to study the effects of using different languages on the number of bug fixing commits while controlling for factors such as project age, project size, team size, and the number of commits. Their findings show that in general implementing a project with more languages has a significant effect on project quality, as it increases defect proneness. Moreover, they find specific languages that are statistically significantly more defect prone when they are used in a multi-language setting. These languages are popular languages like C++, Objective-C, and Java.

6.3 Research Methodology

The overarching goal of this paper is to investigate the use of Design By Contract (DBC) in open source projects. More specifically, we aim to identify how often different types of contracts are used as well as the use cases in which contracts are specified. To this end, we develop a set of 6 parsers to statically analyze source code and detect contract specifications.

To derive the use cases in which contracts are used we took the following approach. First, we select a random number of contracts from each project. To preserve a lower bound in the selection process, we made sure to select at least 10 contracts from each project. We then analyze the source code manually to understand the context and rationale for specifying the contracts. We identify a category for each contract as we analyze it. These categories are not predefined, but rather formed as we comprehend the context and rationale behind the usage of the contracts. At the end, we revise the identified categories and if the categories are too fine-grained, we may combine a number of categories to form larger groups of contracts. To extend our perspective and minimize the risk of making mistakes, we ask five independent (non-author) developers to double check our analysis. Each of the developers has at least 5 years

of professional experience in programming with either Java, or Python, or C++.

In addition, we aim to investigate the relation between using contracts and occurrence of defects. To that end, we develop a set of three commit log parsers for Java, C++, and Python. Similar to Casalnuovo et al. [78] and Kochhar and Lo, [144] for each project, we extract the commit logs using *git log -U1 -W*. The *-U1* argument is used to get the commit patches and *-W* is used to get the source contexts in which the patches were provided. After extracting the Git commit logs, we filter the fixing commits, using the same heuristics that were also used by Casalnuovo et al. [78] and Kochhar and Lo, [144], namely: "fix", "issue", "bug", "defect", "incorrect", "error", "fault", "mistake", and "flaw". We then use the commit log parsers to record an account of every method that is changed in the fixing patches.

To analyze the relation between using contracts and occurrence of defects, we use Poisson regression analysis [198], as opposed to Casalnuovo et al. [78] and Kochhar and Lo, [144] who use Hurdle regression model to analyze relation between developer experience, use of assertions, and frequency of defect occurrences. Hurdle regression analysis is used when the minimum value in the dependent variable is 0. Therefore, Hurdle analysis has two components: Hurdle and Count. Hurdle component measures the effect of overcoming the hurdle, which is 0. On the other hand, the count component measures the effect of going from a non-zero value to another non-zero value.

In our case, the minimum value of the defect occurrence is 1 because all methods were recorded from bug fixing commits where the methods were updated. As a result, we use Poisson regression which is similar to regular multiple regression analysis, which is used to model observed counts. Therefore, the possible values of the dependent variable are non-negative integers in this analysis. In our case, the dependent variable is the number of times a method is updated in the fix patches. The independent variable is the number of contracts used in the methods.

We devise the research questions presented in Section 6.1. In what follows, we further describe our approach to developing the contract and commit log parsers.

6.3.1 Automated Contract Detection

Dataset Collection. According to The State of Octoverse [16, 17], Java, C++ and Python are among the ten most popular languages used in Github. At the same time, these languages are popular instances of strongly typed object oriented, semi object oriented, and high-level interpreted programming languages, respectively, which sup-

port the use of DBC. Therefore, we collected 124 projects which are written in these languages from Github.

Table 6.1: Strings used to search for projects

Library	String
Cofoja	“com.google.java.contract”
Valid4j	“org.valid4j”
JML	“org.jmlspecs.models”
Icontract	“import icontract”
Pycontracts	“from contracts import”
Boost.contract	“boost::contract::check”

Table 6.2: Statistics of the projects

Language	Library	Projects	Files	KLOC
Java	Cofoja	21	6340	767,809
Java	Valid4j	15	1353	72,900
Java	JML	8	15171	1376,265
C++	Boost.contract	48	389146	54239,413
Python	Icontract	9	408	49,010
Python	Pycontracts	23	12787	1839,081

For each language we selected popular libraries that support DBC, namely: JML [31, 149], Valid4j [37], and Cofoja [23] for Java projects, Boost.contract [20] for C++ projects, Icontract [27, 146], and Pycontracts [34] for Python projects. Next, in order to collect the projects using these libraries, we used Github explore [26]. To this end, we first familiarized ourselves with the use of these libraries through their official tutorials. We identified specific key strings, presented in Table 6.1, which are used within source code to import the aforementioned libraries. Therefore, using Github explore, we used the identified strings to search for projects that use a certain library. We then manually investigated the repositories. As long as the contracts were used in the source code and not only in the test code, we included the projects. Later on, contract parsers filter the test files in the projects in order to prevent combining test case assertions with source code contracts in the evaluation. Thereby, we collected 124 projects. Table 6.2 presents the statistics of the collected projects, which are produced by CLOC [19, 22].

Listing 6.1: An example of Cofoja contract specifications.

```
import com.google.java.contract.Ensures;
import com.google.java.contract.Invariant;
import com.google.java.contract.Requires;

@Invariant({ "elements != null", "isEmpty() || top() != null" }) // (1)
public class CofojaStack<T> {

  private final LinkedList<T> elements = new LinkedList<T>();

  @Requires("o != null") // (2)
  @Ensures({ "!isEmpty()", "top() == o" }) // (3)
  public void push(T o) {
    elements.add(o);
  }
 ... }
```

Developing Contract Analyzers. Each of the DBC libraries provide unique syntax for specifying the contracts. Using Cofoja, to define class invariants *@Invariants* are used in the source code. Preconditions and postconditions are specified through *@Requires* and *@Ensures*, respectively. To analyze these contracts, every time these tags are included in a source code line, the Cofoja parser expects to arrive at a class or method declaration line. Therefore, every successive line would be checked until a line where a method declaration is defined arrives. Once the line is found, all the predicates defined within the contract blocks are summed and an account of file names, class names, method names, and the identified contracts is recorded. Listing 6.1 is an example from [25] which illustrates how Cofoja contracts are specified.

On the other hand, using Valid4j, to define preconditions and postconditions, *require* and *ensure* blocks are specified within the method declarations, respectively. In addition, for error handling purposes, *validate* blocks are specified, and unreachable code is specified using *neverGetHere* checks. Therefore, Valid4j parser first identifies method declarations. Once a method declaration is found, the parser checks every successive line in order to identify the contracts. Once end of a method is reached, an account of the file name, class name, method names, and identified contracts are recorded. Listing 6.2 is an example [38] of specifying a precondition for a client method that invokes the *Country* constructor.

Using JML, several different types of contracts can be specified, namely: *requires*,

ensures, *signals*, *invariant*, *non_null*, *pure*, etc. Similar to Cofoja, JML contracts are specified before method declarations begin. Therefore the JML parser takes the same approach as earlier presented for Cofoja. Thus, the JML parser first looks for JML tags, and if any of them is detected, then the following method together with an account of the detected contracts are recorded. Listing 6.3 shows an example [30] of specifying jml contracts.

Listing 6.2: An example of precondition specification using Valid4j.

```
public class Country {

  // Use contract to specify that it is the _clients_ responsibility
  // to make sure only valid country codes are given to the constructor.
  // Invoking this constructor with an invalid country code is considered
  // to be a programming error on the clients part.
  public Country(String code) {
    require(isValidCountryCode(code));
    //
  }
 ...}
```

Listing 6.3: An example of contract specification using JML.

```
/*@ protected normal_behavior
    @  assignable size, theStack;
    @  assignable_redundantly theItems, nextFree;
    @  ensures nextFree == 0;
    @*/
  public BoundedStackImplementation( )
  {
    theItems = new Object[MAX_STACK_ITEMS];
    nextFree = 0;
  }
```

Class invariants, preconditions, postconditions, exception guarantees, and old value copies can be specified as contracts, using Boost.contract [21]. Listing 6.6 is an example [21] that shows how these contracts are specified using Boost.contract.

As Listing 6.6 illustrates, Class invariants are declared using the *void invariant() const* method declaration. Therefore, when the Boost parser identifies the invariant declar-

ations, it will expect the successive lines to include assertions. The parser continues to sum the number of assertions until it reaches the end of the method declaration. As for the rest of the contracts, since they are embedded within the method declarations, the Boost parser detects them in a similar way as the Valid4j parser. Thus, every time a method declaration is detected, Boost parser counts the embedded assertions, if there is any.

Using Icontract, it is possible to specify preconditions, postconditions, invariants, and snapshots, using *@icontract.require*, *@icontract.ensure*, *@icontract.invariant*, and *@icontract.snapshot*, respectively.

@icontract.snapshot are similar to *old* contracts using Boost.contract which record old values of the arguments before state transitions. Therefore, this type of contract can be used to verify the state transitions of arguments [27]. Listing 6.5 shows an example of using *@icontract.snapshot* and *@icontract.ensure* contracts.

Preconditions and postconditions can be specified in three ways, using Pycontract. *@contract* decorators can be used to embed the contract specifications. Therefore, every time the Pycontract parser detects a *@contract* decorator, it counts the number of predicates that are defined within the *@contract* block. In addition, if annotations are used, using Python 3, preconditions and postconditions can also be defined within the method signatures. Therefore, if the Pycontract parser detects a *@contract* tag and Python 3 annotations, it looks for possible contract specifications within the method signatures. Otherwise, if the parser detects a *@contract* tag and no Python 3 annotation is detected, then the parser looks for docstrings with *:type:* and *:rtype:* tags which are used to specify preconditions and postconditions. Listing 6.4 illustrates three example of how to specify contracts, using Pycontracts.

Listing 6.4: Three examples of contract specification using Pycontract.

```
### using an @contract decorator
@contract(a='int,>0', b='list[N],N>0', returns='list[N]')
def my_function(a, b):
    ...
### using an @contract tag and Python 3 annotations
@contract
def my_function(a : 'int,>0', b : 'list[N],N>0') -> 'list[N]':
    # Requires b to be a nonempty list, and the return
    # value to have the same length.
    ...

### using an @contract tag and docstrings
```

```
@contract
def my_function(a, b):
    """ Function description.
        :type a: int,>0
        :type b: list [N],N>0
        :rtype: list [N]
    """
    ...
```

Listing 6.5: An example of contract specification using Icontract.

```
>>> @icontract.snapshot(lambda lst: lst[:])
... @icontract.ensure(lambda OLD, lst, value: lst == OLD.lst + [value])
... def some_func(lst: List[int], value: int) -> None:
...     lst.append(value)
...     lst.append(1984) # bug
```

6.3.2 Parsing Commit Logs

Listing 6.7 illustrates an excerpt from git commit log from the library-manager [32] project, which was generated by the *git log -U1 -W* command. As the listing shows, a git diff that is included in a commit is indicated by the line that starts with *diff –git*. When this line is detected, the parsers record the file name that is referred to in this line. After the commit index, a relative number of changes are indicated by + or − symbols next to the file name. A pair of @@ opening symbols indicate chunk headers where the changed lines are shown. Additional context information such as method names may be provided after the closing @@ symbols, which to a large extent depends on the programming language used in the project and if git internal configurations support the language.

Listing 6.7: Excerpt of a sample git commit log.

```
diff --git a/src/main/java/com/mykosoft/librarymanager/options
/common/BookSelectingById.java
b/src/main/java/com/mykosoft/librarymanager/options/common
/BookSelectingById.java
index 27feeac..a905857 100644
```

Listing 6.6: An example of contract specification using Boost.contract.

```
void invariant() const { // Checked in AND with base class invariants.
    BOOST_CONTRACT_ASSERT(size() <= capacity());
}

 virtual void push_back(T const& value,
        boost::contract::virtual_* v = 0) /* override */ { // For virtuals.
    boost::contract::old_ptr<unsigned> old_size =
            BOOST_CONTRACT_OLDOF(v, size()); // Old values for virtuals.
    boost::contract::check c = boost::contract::public_function< // For overrides.
            override_push_back>(v, &vector::push_back, this, value)
        .precondition([&] { // Checked in OR with base preconditions.
            BOOST_CONTRACT_ASSERT(size() < max_size());
})
        .postcondition([&] { // Checked in AND with base postconditions.
            BOOST_CONTRACT_ASSERT(size() == *old_size + 1);
});
vect_.push_back(value);

}
```

```
--- a/src/main/java/com/mykosoft/librarymanager/options/
common/BookSelectingById.java
+++ b/src/main/java/com/mykosoft/librarymanager/options/
common/BookSelectingById.java
@@ -10,28 +10,27 @@
 public class BookSelectingById implements BookSelectingStrategy {
     private static ConsoleReader reader = new ConsoleReader();

     @Override
     public Book selectBookFromCollection(Collection<Book> booksByTitle)
         throws IOException{
         Book book = null;

         if (booksByTitle.size() > 1) {
             System.out.println("Multiple books exist with that title!");
             ...

         } else {
-            reader.readLine("One book found");
-            System.out.println(book);
             book = booksByTitle.iterator().next();
+            reader.readLine("One book found\n" + book.toString());
         }

         return book;
     }
 }
```

The context of changes begins after the chunk headers. In Listing 6.7, this is where the *BookSelectingById* class is declared. Commit parsers record class names at this point depending on the programming language being analyzed. For example, Python scripts may or may not define classes. From this point onwards, the parsers analyze each successive line until a method declaration arrives. Once a method declaration is detected, the method name and method signature are temporarily recorded. If inside the method, lines indicate changes (by starting with + or −), then the recorded method name and signatures are stored permanently. The parsers continue analyzing the lines and looking for changed methods until the next patch (*diff –git*) or the next commit is detected.

Listing 6.8 is another example of an excerpt from git commit log from the Mapry project [28] which is developed in Python. As the listing shows, the initial lines which indicate the *diff –git*, commit index, and header chunk follow the same format. In this case, the name of the method where the changes were made is also indicate after the closing @@. However, to detect the defective method, parsing Python syntax is required. This is why we developed three git commit parsers for Java, C++, and Python.

In addition, prior to detecting defective methods, merge commits are filtered. In addition, test files are excluded from the analysis in order to maintain the focus on the use of contracts in source code.

Listing 6.8: Another excerpt of a sample git commit log.

```
 diff --git a/docs/source/conf.py b/docs/source/conf.py
index 787f476..086ca83 100644
--- a/docs/source/conf.py
+++ b/docs/source/conf.py
@@ -40,11 +40,13 @@ mapry_meta.__version__
 # Add any Sphinx extension module names here, as strings. They can be
 # extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
 # ones.
 extensions = [
     'sphinx.ext.autodoc',
     'sphinx.ext.doctest',
     'sphinx_autodoc_typehints',
-    'sphinx_icontract'
+    'sphinx_icontract',
+    'sphinx.ext.autosectionlabel'
 ]
+autosectionlabel_prefix_document = True

 # Add any paths that contain templates here, relative to this directory.
```

6.4 Results

In this paper, we aim to investigate the use of contracts in open source projects which are developed in popular programming languages. In this section, we present the

results of our empirical analysis on the use of contracts in open source Java, C++, and Python programs. Thereby, we answer the three research questions presented in Section 6.1.

6.4.1 RQ_1. How often are different types of contracts used?

To answer **RQ_1**, we used the set of parsers we developed in Python in order to automatically detect the specified contracts in Java, C++, and Python projects. As Table 6.3 shows, 57% of the contracts are preconditions in Cofoja projects while 36% are postconditions, and 7% are invariants. 59% of the contracts are preconditions in Valid4j, while 23% are exception guarantees, and 18% are postconditions.

In JML, 35% of the contracts are generic postconditions, while 24% are generic preconditions. In addition, specific contracts are also used as the following: 13% of the contracts in JML are none_null checks, 11% are assignable contracts, 7% are invariants, and 2% are exception signals.

In C++ projects, 46% of the contracts are postconditions, while 29% are preconditions, 18% are invariants, 4% indicate old parameter values, and 2% are exception guarantees.

In Icontract, 65% of the contracts are preconditions, 20% are postconditions, and 15% are invariants. Finally, in Pycontracts, 89% of the contracts are preconditions, while 11% are postconditions.

Table 6.3: Percentages of contract use in Java, C++, and Python projects. **Pre.**, **Post.**, **Inv.**, **Exc.**, **Val.**, **Pure**, **Nnull**, **Sig.**, **Assign.**, and **Nget** indicate “precondition”, “postcondition”, “invariant”, “exception”, “non_null”, “signals”, “assignable”, “nevergethere” contracts, respectively. **-** indicates that the type of contract is not supported by the given library.

Library	Pre.	Post.	Inv.	Old	Exc.	Val.	Pure	Nnull	Sig.	Assign.	Nget.
Cofoja	57	36	7	-	-	-	-	-	-	-	-
Valid4j	59	18	-	-	-	23	-	-	-	-	0
JML	24	35	7	-	-	-	8	13	2	11	-
Boost. contract	29	46	18	4	2	-	-	-	-	-	
Icontract	65	20	15	-	-	-	-	-	-	-	-
Py contracts	89	11	-	-	-	-	-	-	-	-	-

6.4.2 RQ_2. For which use cases do developers use contracts?

To answer **RQ_2**, we randomly selected at least 10 contracts from each project (unless there were less than 10 contracts in a project), which are in total 1505 contracts out of 18494 (8%). We performed manual analysis to identify categories for each contract. At the end, if the categories were too fine-grained, we aggregated them to form a larger group, and vice versa, if a category seemed too coarse grained, we further broke it down to more accurate categories.

As a result, we identify five use cases for contracts, namely: checking null conditions, evaluating objects, evaluating data collections, evaluating strings, and evaluating numbers. The evaluations on the data types are further divided into qualitative and quantitative checks on the values. In what follows, we provide examples we identified during the manual analysis.

Listing 6.9 presents four invariants, from the Cofoja-api project [24], which perform semantic checks on a string. In this case, the string under question is a class name which must always be a simple, qualified, and binary name, according to the invariants.

Listing 6.9: Examples of invariants that evaluate string format.

```
@Invariant({
  "isSimpleName(getSimpleName())",
  "isQualifiedName(getQualifiedName())",
  "isQualifiedName(getSemiQualifiedName())",
  "isBinaryName(getBinaryName())"
})
```

In addition, Listing 6.10 presents examples of contracts, from the Streamline project [36], which provide quantitative and qualitative checks on two lists, `joinPointA` and `joinPointsB`. According to the contracts, the lengths of the lists must be equal while none of the lists can have duplicates. Furthermore, both lists must be sorted, while each member of the lists must be within certain bounds.

Listing 6.10: Examples of qualitative and quantitative checks on lists.

```
  public void validateStreamlineConfiguration(
          Integer[] jointPointsA, Integer[] jointPointsB, Integer capacityA,
               Integer capacityB) {
     valid4jValidator.validate(jointPointsA.length == jointPointsB.length,
               "Number of point pairs must be mutually equal");
```

```
    valid4jValidator.validate(ifNoRepetitions(jointPointsA, jointPointsB),
            "None joint points array must contain duplicates");
    valid4jValidator.validate(ifSorted(jointPointsA, jointPointsB),
            "Joint points within arrays must be sorted in increasing order");
    valid4jValidator.validate(ifPointsAreWithinBounds(joint
             PointsA, jointPointsB, capacityA, capacityB),
            "Joint point must be within capacity bounds");
}
```

Furthermore, Listing 6.11, illustrates contracts, from the MiniUrl project [33], which perform null checks, string semantics checks, and boundary checks on objects. In this case, the contracts require that `originalUrl` and `hashedUrl` arguments are not null, not can `hashedUrl` be an empty string. In addition, the `ttl` which is an immutable instance of `Duration`, cannot be null, 0, or negative.

Listing 6.11: Examples of preconditions and postconditions which perform null checks and boundary checks, respectively.

```
//Proper domain validation when instantiating
public ShortenedUrl(URI originalUrl,
                    String hashedUrl,
                    Duration ttl) {
    this.originalUrl = require(originalUrl, notNullValue());
    this.hashedUrl = require(hashedUrl, notEmptyString());
    this.ttl = require(ttl, notNullValue());
    ensure(!(ttl.isZero() && ttl.isNegative())); //Ensure ttl is positive
}
```

Finally, Listing 6.12, from the INF3143_TP2 project [29], illustrates preconditions which perform semantic checks on an object called `target`. In this case, `target` represents a player, which can neither be null, nor can it be the same as the current instance of a player. In addition, `target` must be still alive so it can be attacked by the current player.

Listing 6.12: An example of preconditions which perform semantic checks on an Object.

```
@Requires({"target != null", // A target can not be null.
    "target != this", // A Player can not attack himself.
    "isAlive()" }) // A Player can not attack if he is dead.
```

```
// The attacking player wins.
@Ensures("getXp() > old(getXp())")
public void attack(Player target) {
    int dmg = 20 + this.getStrength() - target.getEndurance();
    Logger.getLogger().log("Player " + this + " attacks " + target);
    target.hurt(dmg);
    this.gainXp(dmg * 2);
}
```

6.4.3 RQ3. Does the use of contracts relate to occurrence of defects?

To answer **RQ3**, we performed Poisson regression analysis in order to identify if there is a relation between the use of contracts and occurrence of defects in a method. Table 6.4 presents the results of the regression analysis. According to the table, all estimates are negative, which means that there is a negative relation between the use of contracts and the frequency of defect occurrence. In other words, the results of the regression analysis show that defect density is lower when contracts are used in projects. In addition, as the table shows, all p values are less than 0.05, which means the observed relation is statistically significant.

In addition, we further performed regression analysis for each project separately in order to observe the results of each project independently. The same observation holds for the majority of the projects in Java, C++, and Python. For 2% of the projects which use Cofoja, 3% of the projects which use Valid4j, 7% of the projects which use Boost.contract, and 5% of the projects which use Pycontracts, we did not observe statistically significant effects for using contracts on defect occurrence.

6.5 Discussion

6.5.1 Preference for Different Contracts

Estler et al. [98] investigated 21 contract-equipped Eiffel, C#, and Java projects to identify which types of contracts are used more often, and how contracts evolve over

Table 6.4: The results from Poisson regression analysis to analyze the relation between the number of contracts and frequency of defect occurrence.*** indicates that $p < 0.001$. * indicates that $p < 0.05$.

Library	Coefficients:	Std. Error	Z value	Pr(>|z|)
Cofoja	-5.473595	0.288201	-18.99	$2e^{-}16$ ***
Valid4j	-0.79738	0.35435	-2.25	0.0244 *
JML	-5.451936	0.407862	-13.37	$2e^{-}16$ ***
Boost	-3.61906	0.15130	-23.92	$2e^{-}16$ ***
Icontract	-4.43053	0.46262	-9.577	$2e^{-}16$ ***
Pycontracts	-3.706112	0.226755	-16.34	$2e^{-}16$ ***

time. They report that contracts are stable over time and that they did not observe strong preference for different contracts.

The results of our analysis on 124 Java, C++, and Python projects show, in average, preconditions form the majority of the contracts used in projects which use Cofoja, Valid4j, Icontract and Pycontracts. However, in the case of JML, in average, generic postconditions were used more often compared to generic preconditions. At the same time, JML provides more specific contracts, namely, *pure*, *non_null*, *signal*, and *assignable*, which together, in average, form 33% of the JML contracts that are used in Java projects.

In the case of Boost.contract, in average, postconditions form the majority of the contracts used in C++ projects. Finally, in the case of Icontract and Pycontracts, similar trends as Cofoja and Valid4j are observed, meaning, preconditions are, in average, the majority of the contracts used in Python projects.

Therefore, we observe that our results contradict the results reported by Estler et al. [98] in that we observe strong preferences for using generic preconditions in several cases. Future research may investigate developers' underlying reasons for preferring different types of contracts for specification purposes.

6.5.2 Automated Semantic Analysis of Contracts

We observe that JML, Boost.contract, and Valid4j provide more specific constructs to indicate specific conditions. On the other hand, Cofoja, Icontract, and Pycontract provide standard constructs to specify generic preconditions, postconditions, and invariants.

Using specific contracts facilitates means to develop automated approaches to detect-

ing the semantic use cases of contracts. However, currently there is little support for fine-grain contract specification.

Furthermore, in the case of JML, even though various specific constructs are available for contract specification, in average, generic postconditions still form the majority of the contracts that are used in open source projects. Future work may pursue further advancements in development and analysis of using fine-grained constructs for contracts specifications.

6.5.3 Effect of Using Contracts

As Dietrich et al. [94] highlight, projects which use contracts continue to do so, as a result, they will expand the use of contracts as they evolve. However, in general, contracts are used less than expected. On the other hand, according to the results from Poisson regression analysis, we observe that using contracts does impact the frequency of defect occurrences in open source projects. Despite the advantages of contracts, the use of contracts does not progress fast. Future research may investigate the perception of developers in this regard.

6.6 Threats to Validity

In what follows, we present the threats to internal and external validities, respectively.

6.6.1 Threats to Internal Validity

In order to minimize the risk of having faults in the developed Python code, we reviewed the source code and wrote test cases. However, with this approach, it is not possible to entirely guarantee absence of defects. As mentioned in Section 6.1, we provide a replication package and intend to make all contributions publicly available in the future. We believe taking this approach and fostering openness increases the possibilities to identify potential faults.

With reards to defect identification from commit logs, we used the same approach as Casalnuovo et al. [78] and Kochhar and Lo, [144], instead of using defect databases. This approach entails false positives and or false negatives, similar to links to defects in defect databases which may also contain errors according to Bird et al. [66]. On the other hand, according to Bissyandé et al. [67], not all projects use issue trackers.

Also, not all bugs are recorded in issue trackers. In our case, a defect may appear in several bug fixing commits. In addition, if bug fixing commits are not sufficiently accurate, bug fixing commits may be missed.

During the manual analysis process, we asked feedback from five independent developers who had several years of professional experience in programming. Incorporating the feedback from professional developers helped us minimize the risk of making mistakes in our manual analysis.

6.6.2 Generalizability of Findings

We used Github to collect the projects. Github is a popular platform which hosts over 96 million repositories [35]. Over 31 million developers contribute to projects which are hosted on Github. In addition, we studied three different programming languages and formed a large corpus of projects.

Moreover, we used statistical regression model to perform analysis on the relation of defect occurrence and contracts use. Since the results of the regression analysis are statistically significant, it is possible to extrapolate the observations reported in this paper.

However, while we tried to minimize the threats to generalizability, we cannot argue that our results generalize to industrial closed-source software. Therefore, future studies may replicate this study in closed-source projects and compare the outcomes.

6.7 Conclusion

Delivering robust and reliable software is a pressing demand for software developers. These qualities matter even more when software components are developed to be reused in many applications. To support development of reliable software, Meyer [162] [163] provides pragmatic techniques and guidelines based on the concept of Design By Contract (DBC).

DBC is based on the idea that software components collaborate with each other based on mutual expectations and benefits. Thus is a client component uses services of a supplier component, the client needs to guarantee preconditions of the supplier. The supplier in return guarantees certain conditions at the time of delivering the services. Thus, contracts are a kind of formal specifications that can be used for different purposes such as automated testing, and static and dynamic verifications.

Despite the advantages of using contracts, contracts are used less than expected according to Dietrich et al. [94]. On the other hand, assertions, exceptions and other built-in features of programming languages are used as lightweight contracts. Therefore, the use of assertions and other lightweight contracts have been studied in depth by Kudrjavets et al. [147], Casalnuovo et al. [78], and Kochhar and Lo [144]. However, the use of contracts has been underexplored to the best of our knowledge. In this paper, we aim to investigate the use of contracts in open source projects. In addition, we aim to analyze the relation between using contracts and frequency of defect occurrences. Therefore, we studied 124 Java, C++, and Python projects. We developed a set of contract parsers which we used to parse contract-equipped source code. The results of automated contract detection show in most cases, except for C++ projects, the use of preconditions, in average, forms the majority of the contract specifications. We derived five categories of contracts which are: null checks, checks on objects, checks on data collections, checks on strings, checks on numbers, Moreover, we use Poisson regression analysis to identify the relation between the use of contracts and defect occurrences. The results of the regression analysis confirms there is a statistically significant negative relation between the use of contracts and defect occurrences. Thus, when methods use contracts, the rate of defect occurrences becomes smaller.

7 Dutch Summary

Tegenwoordig vertrouwen we op verschillende manieren op computersystemen. Toch zijn deze systemen gevoelig voor falen. Deze mislukkingen kunnen levens verstoren, doden veroorzaken en miljarden dollars kosten. Daarom spelen testen en verificatie van software een cruciale rol bij het voorkomen van dergelijke catastrofale fouten. Omdat softwaretests en verificatieactiviteiten kostbaar en arbeidsintensief zijn, is veel moeite gedaan om zoveel mogelijk activiteiten op deze gebieden te automatiseren. In dit proefschrift is het overkoepelende doel om verschillende manieren te onderzoeken om geautomatiseerde software-foutopsporing te vergemakkelijken.

We presenteren *EvoCrash*, een zoekgebaseerde benadering van geautomatiseerde crash-reproductie. *EvoCrash* past een genetisch algoritme toe om te zoeken naar een test-case die een softwarecrash reproduceert. We hebben een grootschalige evaluatie uit-gevoerd om de prestaties van de EvoCrash-aanpak te beoordelen en de gebieden te identificeren waar verdere verbetering nodig is.

Verder introduceren we de IMaChecker-aanpak, die Github bug repositories verkent, met behulp van Github API's. *IMaChecker* analyseert bovendien bugrapporten en iden-tificeert welke elementen (bijv. Reproductiestappen) erin zijn opgenomen. Met behulp van statistische tests identificeert *IMaChecker* de impact van verschillende elementen van het bugrapport op de tijden voor het oplossen van de bugs.

Ten slotte ontwikkelen we statische analyzers die het gebruik van programmacon-tracten in open source programma's detecteren die zijn ontwikkeld in Java, C ++ en Python. We ontwikkelen verder parsers om fixing commits te identificeren bij de hele commit historie. Met behulp van Poisson-regressietests laten we zien dat er een

negatieve correlatie bestaat tussen het gebruik van contracten en het optreden van bugs. We tonen dus een manier om bugs te voorkomen door programmacontracten te gebruiken.

8 English Summary

Today we rely on computer systems in numerous ways. Yet, these systems are susceptible to failure. These failures may disrupt lives, cause deaths, and cost billions of dollars. Thus, software testing and verification play paramount roles in attempting to prevent such catastrophic failures. Since software testing and verification activities are costly and labor-intensive, much effort has been put into automating as many activities in these areas as possible. In this thesis, the overarching goal is to investigate different means to facilitate automated software debugging.

We present *EvoCrash*, which is a search-based approach to automated crash reproduction. *EvoCrash* applies a genetic algorithm to search for a test case that reproduces a software crash. We performed a large-scale evaluation to asses the performance of the EvoCrash approach and identify the areas where further improvement is needed.

Furthermore, we introduce the *IMaChecker* approach, which mines Github bug repositories, using Github APIs. In addition, *IMaChecker* parses bug reports and identifies which elements (e.g. reproducing steps) are included in them. Using statistical tests, *IMaChecker* identifies the impact of different bug report elements on bug resolution times.

Finally, we develop static analyzers which detect the use of program contracts in open source programs which are developed in Java, C++, and Python. In addition, we develop parsers to identify fixing commits among the entire commit histories. Using Poisson regression tests, we show there is a negative correlation between the use of contracts and bug occurrences. Thus, we show one way to avoid bugs is to use program contracts.

9 About the Author

Mozhan Soltani, born in 1989, received her B.Sc. degree in Software Engineering and Management from Gothenburg University, Gothenburg, Sweden, in 2014. She received her M.Sc. in Software Engineering from Gothenburg University, Gothenburg, Sweden, in 2016. In 2016, Mozhan Soltani moved to the Netherlands to do her PhD research on the use of automated test generation for software debugging at the Software Engineering Research Group (SERG), at Delft University of Technology. In 2019, she moved to Leiden University to complete the PhD research on exploring means to facilitate software debugging.

Bibliography

[1] https://github.com/netty/netty/issues/8285, accessed on 2019-06-20.

[2] https://github.com/axios/axios/issues/376, accessed on 2019-06-20.

[3] https://github.com/axios/axios/issues/246, accessed on 2019-06-20.

[4] https://github.com/apache/dubbo/issues/1500, accessed on 2019-06-20.

[5] https://github.com/apache/dubbo/issues/3236, accessed on 2019-06-20.

[6] https://github.com/activeadmin/activeadmin/issues/2623, accessed on 2019-06-20.

[7] https://github.com/activeadmin/activeadmin/issues/3185, accessed on 2019-06-20.

[8] https://github.com/activeadmin/activeadmin/issues/4650, accessed on 2019-06-20.

[9] https://developer.github.com/v3/issues/, accessed on 2015-05-01.

[10] https://github.com/angular/angular.js/issues/16697, accessed on 2019-06-20.

[11] https://github.com/ageitgey/face_recognition/issues/854, accessed on 2019-06-20.

[12] https://github.com/airbnb/lottie-android/issues/1202, accessed on 2019-06-20.

[13] https://github.com/barryvdh/laravel-debugbar/issues/237, accessed on 2019-06-20.

[14] https://github.com/activeadmin/activeadmin/issues/4250, accessed on 2019-06-20.

[15] The state of the octoverse. https://octoverse.github.com/, accessed on 2019-05-01.

[16] The state of the octoverse 2017. https://octoverse.github.com/2017/, accessed on 2019-05-01.

[17] The state of the octoverse: top programming languages of 2018. https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/, accessed on 2019-05-01.

[18] UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey.html`, 2017. [Online; accessed 24-July-2017].

[19] Aldanial, 2019. https://github.com/AlDanial/cloc, Accessed: 2019-08-20.

[20] Boost, 2019. https://www.boost.org/, accessed on 2019-07-01.

[21] Boost.contract example, 2019. https://www.boost.org/doc/libs/1_67_0/libs/contract/doc/html/boost/contract/public_functio_idp69202896.html, Accessed:
2019-08-20.

[22] Cloc, 2019. http://cloc.sourceforge.net/, Accessed: 2019-08-20.

[23] Cofoja. https://github.com/nhatminhle/cofoja, 2019. accessed on 2019-07-01.

[24] cofoja-api, 2019. https://github.com/wao/cofoja-api, Accessed: 2019-08-22.

[25] Cofoja example, 2019. http://blog.code-cop.org/2018/02/complete-cofoja-setup-example.html, Accessed: 2019-08-20.

[26] Github explore, 2019. https://github.com/explore, Accessed: 2019-07-15.

[27] icontract, 2019. https://pypi.org/project/icontract/, accessed on 2019-07-01.

[28] Inf3143_tp2, 2019. https://github.com/Parquery/mapry, Accessed: 2019-08-22.

[29] Inf3143_tp2, 2019. https://github.com/nic-lovin/INF3143_TP2, Accessed: 2019-08-22.

[30] Jml example, 2019. http://www.eecs.ucf.edu/ leavens/JML-release/org/jmlspecs/samples/stacks/, Accessed: 2019-08-20.

[31] Jml home page: The java modeling language (jml), 2019. http://www.eecs.ucf.edu/ leavens/JML/index.shtml, accessed on 2019-07-01.

[32] library-manager, 2019. https://github.com/openminded-oscar/library-manager, Accessed: 2019-08-22.

[33] Miniurl, 2019. https://github.com/gunbuster135/MiniUrl, Accessed: 2019-08-22.

[34] Pycontracts, 2019. https://andreacensi.github.io/contracts/, accessed on 2019-07-01.

[35] The state of the octoverse, 2019. https://octoverse.github.com/, accessed on 2019-05-01.

[36] streamline, 2019. https://github.com/denipotapov/streamline, Accessed: 2019-08-22.

[37] Valid4j, 2019. http://www.valid4j.org/, accessed on 2019-07-01.

[38] Valid4j example, 2019. http://www.valid4j.org/concepts.html, Accessed: 2019-08-20.

[39] S. Adee. Bad bugs: The worst disasters caused by software fails, 2019.

[40] S. Adolph, W. Hall, and P. Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[41] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361, March 2013.

[42] N. M. Albunian. Diversity in search-based unit test suite generation. In *Int'l Symposium on Search Based Software Engineering*, pages 183–189. Springer, 2017.

[43] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin. Deploying Search Based Software Engineering with Sapienz at Face-

book. In *Search-Based Software Engineering. SSBSE 2018.*, volume 11036 of *LNCS*. Springer, 2018.

[44] A. Ang, A. Perez, A. van Deursen, and R. Abreu. *Revisiting the Practical Use of Automated Software Fault Localization Techniques*. IEEE, United States, 2017.

[45] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[46] Apache. Ant. `http://ant.apache.org/`, 2017. [Online; accessed 25-January-2018].

[47] Apache. Commons Collections. `https://commons.apache.org/proper/commons-collections/`, 2017. [Online; accessed 25-January-2018].

[48] Apache. Log4j. `https://logging.apache.org/log4j/2.x/`, 2017. [Online; accessed 25-January-2018].

[49] A. Arcuri. RESTful API Automated Test Case Generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 9–20. IEEE, jul 2017.

[50] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[51] A. Arcuri and G. Fraser. On Parameter Tuning in Search Based Software Engineering. In *Population English Edition*, pages 33–47. 2011.

[52] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[53] A. Arcuri and G. Fraser. Java enterprise edition support in search-based junit test generation. In *International Symposium on Search Based Software Engineering*, pages 3–17. Springer, 2016.

[54] A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 79–90, Vasteras, Sweden, 2014. ACM Press.

[55] A. Arcuri, G. Fraser, and J. P. Galeotti. Generating tcp/udp network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 155–165. ACM, 2015.

[56] A. Arcuri, G. Fraser, and R. Just. Private api access and functional mocking in automated unit test generation. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 126–137, Tokyo, Japan, 2017. IEEE, IEEE Computer Society.

[57] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 49–60. ACM, 2010.

[58] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.

[59] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 53–62. IEEE Computer Society, 2011.

[60] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.

[61] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 108–118, Boston, Massachusetts, USA, 2004. ACM, ACM.

[62] K. L. Barriball and A. While. Collecting data using a semi-structured interview: a discussion paper. *Journal of Advanced Nursing-Institutional Subscription*, 19(2):328–335, 1994.

[63] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[64] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press.

[65] F. A. Bianchi, M. Pezzè, and V. Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 705–716, Paderborn, Germany, 2017. ACM, ACM.

[66] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[67] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillere, J. Klein, and Y. Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pages 188–197. IEEE, 2013.

[68] H. Borges, M. T. Valente, A. Hora, and J. Coelho. On the popularity of github applications: A preliminary note. *arXiv preprint arXiv:1507.00604*, 2015.

[69] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[70] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 90–101, New York, NY, USA, 2017. ACM.

[71] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[72] L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh. The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated. *IEEE Software*, 34(5):72–75, 2017.

[73] O. Bühler and J. Wegener. Evolutionary functional testing. *Computers & Operations Research*, 35(10):3144–3160, 2008.

[74] R. P. Buse, C. Sadowski, and W. Weimer. Benefits and barriers of user evaluation in software engineering research. *ACM SIGPLAN Notices*, 46(10):643–656, 2011.

[75] B. Cabral and P. Marques. Exception Handling: A Field Study in Java and .NET. In *ECOOP 2007 âĂŞ Object-Oriented Programming*, volume 4609, pages 151–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[76] Y. Cao, H. Zhang, and S. Ding. Symcrash: Selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 791–802. ACM, 2014.

[77] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 755–766. IEEE Press, 2015.

[78] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 755–766. IEEE Press, 2015.

[79] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *Proceedings of the 34th International Conference on Software Engineering*, pages 452–462. IEEE Press, 2012.

[80] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 121–130. ACM, 2011.

[81] N. Chen and S. Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Tr. on Sw. Eng.*, 41(2):198–220, 2015.

[82] H. Cibulski and A. Yehudai. Regression test selection techniques for test-driven development. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 115–124, March 2011.

[83] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *29th International Conference on Software Engineering (ICSE'07)*, pages 261–270. IEEE, 2007.

[84] R. Coelho, L. Almeida, G. Gousios, A. v. Deursen, and C. Treude. Exception handling bug hazards in android. *Empirical Software Engineering*, pages 1–41, 2016.

[85] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 134–145. IEEE Press, 2015.

[86] R. Coelho, L. Almeida, G. Gousios, A. van Deursen, and C. Treude. Exception handling bug hazards in Android. *Empirical Software Engineering*, 22(3):1264–1304, jun 2017.

[87] C. A. Coello Coello. Constraint-handling techniques used with evolutionary algorithms. In *Proc. of the Genetic and Evolutionary Computation Conference Companion (GECCO Companion)*, pages 563–587. ACM, 2016.

[88] M. Črepinšek, S.-H. Liu, and M. Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 45(3):35, 2013.

[89] J. W. Creswell and J. D. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[90] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, pages 1084–1093. IEEE Press, 2012.

[91] D. De Vaus and D. de Vaus. *Surveys in social research*. Routledge, 2013.

[92] K. Deb. Multi-objective optimization. In *Search Methodologies*, pages 403–449. Springer US, 2014.

[93] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *TEVC*, 6(2):182–197, 2002.

[94] J. Dietrich, D. J. Pearce, K. Jezek, and P. Brada. Contracts in the wild: A study of java programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[95] Dubbo. A high-performance, java based, open source RPC framework. `http://dubbo.io`, 2018. [Online; accessed 25-January-2018].

[96] F. Eichinger, K. Krogmann, R. Klug, and K. Böhm. Software-defect localisation by mining dataflow-enabled call graphs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 425–441. Springer, 2010.

[97] Elastic. Elasticsearch: RESTful, Distributed Search and Analytics. `https://www.elastic.co/products/elasticsearch`, 2018. [Online; accessed 25-January-2018].

[98] H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *International Symposium on Formal Methods*, pages 230–246. Springer, 2014.

[99] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *Proc. Int'l Conf. Software*

Testing Verification and Validation Workshops (ICSTW), pages 178–186. IEEE, 2008.

[100] A. Fink. *The survey handbook*. Sage, 2003.

[101] G. Fraser and A. Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2013.

[102] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 362–369. IEEE, 2013.

[103] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, Feb. 2013.

[104] G. Fraser and A. Arcuri. Automated test generation for java generics. In *International Conference on Software Quality*, pages 185–198, Vienna, Austria, 2014. Springer, Springer.

[105] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[106] G. Fraser, A. Arcuri, and P. McMinn. A memetic algorithm for whole test suite generation. *Journal of Systems and Software*, 103:311–327, 2015.

[107] G. Fraser, J. M. Rojas, J. Campos, and A. Arcuri. E vo s uite at the sbst 2017 tool competition. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*, pages 39–41. IEEE Press, 2017.

[108] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 291–301. ACM, 2013.

[109] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? A controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4):23:1–23:49, 2015.

[110] G. R. Gibbs. Thematic coding and categorizing. *Analyzing qualitative data. London: Sage*, pages 38–56, 2007.

[111] B. G. Glaser and J. Holton. Remodeling grounded theory. In *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, volume 5, 2004.

[112] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[113] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proc. Int'l Conf. on Genetic Algorithms and Their Application*, pages 41–49. L. Erlbaum Associates Inc., 1987.

[114] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 88–99, New York, NY, USA, 2016. ACM.

[115] F. Gordon and A. Arcuri. Evosuite at the sbst 2016 tool competition. In *The 9th International Workshop on SEARCH-BASED SOFTWARE TESTING (SBST)*, 2016.

[116] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[117] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[118] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1359–1366, New York, USA, 2002. Morgan Kaufmann Publishers Inc., Morgan Kaufmann.

[119] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[120] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

[121] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83. ACM, 2007.

[122] M. Harman, P. McMinn, J. De Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2012.

[123] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[124] M. Höst, B. Regnell, and C. Wohlin. Using students as subjectsâĂŤa comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

[125] S. E. Hove and B. Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10–pp. IEEE, 2005.

[126] S. A. Jacob and S. P. Furgerson. Writing interview protocols and conducting interviews: Tips for students new to the field of qualitative research. *The qualitative report*, 17(42):1–10, 2012.

[127] M. Jähne, X. Li, and J. Branke. Evolutionary algorithms and multi-objectivization for the travelling salesman problem. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 595–602. ACM, 2009.

[128] Java Design Patterns. Design patterns implemented in Java. `http://java-design-patterns.com`, 2018. [Online; accessed 25-January-2018].

[129] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object Capture-based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 159–170, New York, NY, USA, 2010. ACM.

[130] JDK. Stack trace has invalid line numbers. `https://bugs.openjdk.java.net/browse/JDK-7024096`, 2016. [Online; accessed 25-January-2018].

[131] C. Jee and T. Macaulay. Top software failures in recent history, 2019.

[132] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 249–258. IEEE, 2008.

[133] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang. Why and how developers fork what from whom in github. *Empirical Software Engineering*, 22(1):547–578, 2017.

[134] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

[135] R. Just, D. Jalali, and M. 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*, pages 437–440, San Jose, CA, USA, 2014. ACM Press.

[136] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella. Sbfr: A search based approach for reproducing failures of programs with grammar based input. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 604–609, Piscataway, NJ, USA, 2013. IEEE Press.

[137] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella. Reproducing field failures for programs with complex grammar-based input. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 163–172, Washington, DC, USA, 2014. IEEE Computer Society.

[138] F. M. Kifetew, A. Panichella, A. De Lucia, R. Oliveto, and P. Tonella. Orthogonal exploration of the search space in evolutionary test case generation. In *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*, pages 257–267. ACM, 2013.

[139] B. A. Kitchenham and S. L. Pfleeger. Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes*, 27(2):20–24, 2002.

[140] J. D. Knowles, R. A. Watson, and D. W. Corne. Reducing local optima in single-objective problems by multi-objectivization. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 269–283. Springer, 2001.

[141] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 353–356. IEEE, 2013.

[142] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *2013 13th International Conference on Quality Software*, pages 103–112. IEEE, 2013.

[143] P. S. Kochhar and D. Lo. Revisiting assert use in github projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 298–307. ACM, 2017.

[144] P. S. Kochhar and D. Lo. Revisiting assert use in github projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 298–307. ACM, 2017.

[145] P. S. Kochhar, D. Wijedasa, and D. Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573. IEEE, 2016.

[146] R. Kramer. icontract-the java/sup tm/design by contract/sup tm/tool. In *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176)*, pages 295–307. IEEE, 1998.

[147] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *2006 17th International Symposium on Software Reliability Engineering*, pages 204–212. IEEE, 2006.

[148] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[149] G. T. Leavens and Y. Cheon. Design by contract with jml, 2006.

[150] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test cases. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE ’07, pages 425–434, New York, NY, USA, 2007. ACM.

[151] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol. On the effectiveness of test extraction without overhead. In *2009 International Conference on Software Testing Verification and Validation*, pages 416–425. IEEE, 2009.

[152] Y. Li and G. Fraser. Bytecode testability transformation. In *International Symposium on Search Based Software Engineering*, pages 237–251, Szeged, Hungary, 2011. Springer, Springer.

[153] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[154] B. Liskov and J. Guttag. *Program development in JAVA: abstraction, specification, and object-oriented design*. Pearson Education, London, England, UK, 2000.

[155] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 643–653, 2014.

[156] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[157] A. Maiga, A. Hamou-Lhadj, M. Nayrolles, K. Koochekian Sabor, and A. Larsson. An empirical study on the handling of crash reports in a large software company: An experience report. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 342–351, Bremen, Germany, sep 2015. IEEE.

[158] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 436–439, Lawrence, KS, USA, 2011. IEEE, IEEE Computer Society.

[159] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2016.

[160] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, jun 2004.

[161] P. McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163, Berlin, Germany, 2011. IEEE, IEEE Computer Society.

[162] B. Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[163] B. Meyer. Building bug-free oo software: An introduction to design by contract. *Availabe at http://archive. eiffel. com/doc/manuals/technology/contract*, 1998.

[164] A. Moghaddam. Coding issues in grounded theory. *Issues in educational research*, 16(1):52–66, 2006.

[165] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing android

application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 33–44, April 2016.

[166] T. Mortensen, R. Fisher, and G. Wines. Students as surrogates for practicing accountants: Further evidence. In *Accounting Forum*, volume 36, pages 251–265. Elsevier, 2012.

[167] M. D. Myers and M. Newman. The qualitative interview in is research: Examining the craft. *Information and organization*, 17(1):2–26, 2007.

[168] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[169] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[170] M. Nayrolles and A. Hamou-Lhadj. BUMPER: A Tool for Coping with Natural Language Searches of Millions of Bugs and Fixes. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 649–652, Suita, Osaka, Japan, mar 2016. IEEE.

[171] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson. Jcharming: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 101–110. IEEE, 2015.

[172] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process*, pages n/a–n/a, 2016. JSME-15-0137.R1.

[173] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process*, 29(3):e1789, mar 2017.

[174] G. News and Media Limited or its affiliated companies. Robot kills worker at Volkswagen plant in Germany, 2015.

[175] Oracle. What's New in JDK 8. `https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`, 2019. Accessed: 2019-05-14.

[176] A. Orso and G. Rothermel. Software testing: a research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, pages 117–132. ACM, 2014.

[177] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[178] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.

[179] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Trans. Software Eng.*, 41(4):358–383, 2015.

[180] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: an empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 547–558, 2016.

[181] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[182] S. Parsa, M. Vahidi-Asl, S. Arabi, and B. Minaei-Bidgoli. Software fault localization using elastic net: A new statistical approach. In *International Conference on Advanced Software Engineering and Its Applications*, pages 127–134. Springer, 2009.

[183] S. L. Pfleeger and B. A. Kitchenham. Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, 26(6):16–18, 2001.

[184] I. S. W. B. Prasetya. *T3, a Combinator-Based Random Testing Tool for Java: Benchmarking*, pages 101–110. Springer International Publishing, Cham, 2014.

[185] I. W. B. Prasetya. T3, a combinator-based random testing tool for java: benchmarking. In *International Workshop on Future Internet Testing*, pages 101–110. Springer, 2013.

[186] W. Prasetya, T. Vos, and A. Baars. Trace-based reflexive testing of oo programs with t2. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 151–160. IEEE, 2008.

[187] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 134–143. IEEE, 1998.

[188] R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 286–293. IEEE, 2012.

[189] J. Roche. Adopting DevOps practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[190] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108. Springer, 2015.

[191] J. M. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 338–349. ACM, 2015.

[192] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, aug 2016.

[193] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893, 2017.

[194] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, pages 114–123. IEEE, 2013.

[195] RxJava. Reactive Extensions for the JVM. `https://github.com/ReactiveX/RxJava`, 2018. [Online; accessed 25-January-2018].

[196] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, 2014.

[197] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121. IEEE, 2010.

[198] S. S. Sharma, D. G. Kleinbaum, and L. L. Kupper. Applied regression analysis and other multivariate methods. 1978.

[199] S. E. Sim, S. Easterbrook, and R. C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 74–83, Portland, Oregon, USA, 2003. IEEE Computer Society.

[200] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *IEEE transactions on software engineering*, 31(9):733–753, 2005.

[201] M. Soltani, A. Panichella, and A. van Deursen. Evolutionary testing for crash reproduction. In *Proceedings of the 9th International Workshop on Search-Based Software Testing - SBST '16*, pages 1–4, 2016.

[202] M. Soltani, A. Panichella, and A. van Deursen. A Guided Genetic Algorithm for Automated Crash Reproduction. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 209–220, Buenos Aires, Argentina, may 2017. IEEE.

[203] M. Soltani, A. Panichella, and A. van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*, pages 209–220. IEEE Press, 2017.

[204] M. Soltani, A. Panichella, and A. van Deursen. Search-Based Crash Reproduction and Its Impact on Debugging. *Software Engineering, IEEE Transactions on*, 2018.

[205] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 158–167, New York, NY, USA, 2000. ACM.

[206] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[207] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[208] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. *SIGARCH Comput. Archit. News*, 38(1):155–166, Mar. 2010.

[209] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[210] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[211] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE, 2007.

[212] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 551–560. ACM, 2011.

[213] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[214] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux. Precise identification of problems for structural test generation. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 611–620, Waikiki, Honolulu , HI, USA, 2011. IEEE, ACM.

[215] J. Xuan, X. Xie, and M. Monperrus. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 910–913, New York, New York, USA, 2015. ACM Press.

[216] XWiki. The Advanced Open Source Enterprise and Application Wiki. `http://www.xwiki.org/`, 2018. [Online; accessed 25-January-2018].

[217] T. Yu, T. S. Zaman, and C. Wang. Descry: reproducing system-level concurrency failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 694–704. ACM, 2017.

[218] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of*

the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 249–265, Berkeley, CA, USA, 2014. USENIX Association.

[219] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 321–334, New York, NY, USA, 2010. ACM.

[220] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[221] A. Zeller. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.

[222] H. Zeng and D. Rine. Estimation of software defects fix effort using neural networks. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, volume 2, pages 20–21. IEEE, 2004.

[223] C. Zhang, J. Yang, D. Yan, S. Yang, and Y. Chen. Automated breakpoint generation for debugging. *JSW*, 8(3):603–616, 2013.

[224] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.