UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2019-79
The Faculty of Science, Technology and Communication

Dissertation

Presented on the 18/12/2019 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Kui Liu
Born on 11th August 1988 in Jiangxi, China

Deep Pattern Mining for Program Repair

Dissertation Defense Committee

Dr. Yves Le Traon, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Tegawendé Bissyandé, Chairman
Assistant Professor, University of Luxembourg, Luxembourg

Dr. Dongsun Kim, Vice Chairman
Quality Assurance Engineer, Furiosa.ai, South Korea

Dr. Jacques Klein, Expert
Associate Professor, University of Luxembourg, Luxembourg

Dr. Andreas Zeller
Professor, Saarland University, Saarbrucken, Germany

Dr. David Lo
Associate Professor, Singapore Management University, Singapore

Abstract

Error-free software is a myth. Debugging thus accounts for a significant portion of software maintenance and absorbs a large part of software cost. In particular, the manual task of fixing bugs is tedious, error-prone and time-consuming. In the last decade, automatic bug-fixing, also referred to as automated program repair (APR) has boomed as a promising endeavor of software engineering towards alleviating developers' burden. Several potentially promising techniques have been proposed making APR an increasingly prominent topic in both the research and practice communities. In production, APR will drastically reduce time-to-fix delays and limit downtime. In a development cycle, APR can help suggest changes to accelerate debugging.

As an emergent domain, however, program repair has many open problems that the community is still exploring. Our work contributes to this momentum on two angles: the repair of programs for functionality bugs, and the repair of programs for method naming issues. The thesis starts with highlighting findings on key empirical studies that we have performed to inform future repair approaches. Then, we focus on template-based program repair scenarios and explore deep learning models for inferring accurate and relevant patterns. Finally, we integrate these patterns into APR pipelines, which yield the state of the art repair tools. The dissertation includes the following contributions:

- *Real-world Patch Study*: Existing APR studies have shown that the state-of-the-art techniques in automated repair tend to generate patches only for a small number of bugs even with quality issues (e.g., incorrect behavior and nonsensical changes). To improve APR techniques, the community should deepen its knowledge on repair actions from real-world patches since most of the techniques rely on patches written by human developers. However, previous investigations on real-world patches are limited to statement level that is not sufficiently fine-grained to build this knowledge. This dissertation starts with deepening this knowledge via a systematic and fine-grained study of real-world Java program bug fixes.
- *Fault Localization Impact*: Existing test-suite-based APR systems are highly dependent on the performance of the fault localization (FL) technique that is the process of the widely studied APR pipeline. However, APR systems generally focus on the patch generation, but tend to use similar but different strategies for fault localization. To assess the impact of FL on APR, we identify and investigate a practical bias caused by the FL step in a repair pipeline. We propose to highlight the different FL configurations used in the literature, and their impact on APR systems when applied to the real bugs. Then, we explore the performance variations that can be achieved by "tweaking" the FL step.
- *Fix Pattern Mining*: Fix patterns (a.k.a. fix templates) have been studied in various APR scenarios. Particularly, fix patterns have been widely used in different APR systems. To date, fix pattern mining is mainly studied in three ways: manually summarization, transformation inferring and code change action statistics. In this dissertation, we explore mining fix patterns for static bugs leveraging deep learning and clustering algorithms.
- AVATAR: *Fix pattern based patch generation* is a promising direction in the APR community. Notably, it has been demonstrated to produce more acceptable and correct patches than the patches obtained with mutation operators through genetic programming. The performance of fix pattern based APR systems, however, depends on the fix ingredients mined from commit changes

in development histories. Unfortunately, collecting a reliable set of bug fixes in repositories can be challenging. We propose to investigate the possibility in an APR scenario of leveraging code changes that address violations by static bug detection tools. To that end, we build the Avatar APR system, which exploits fix patterns of static analysis violations as ingredients for patch generation.

- *TBar*: Fix patterns are widely used in patch generation of APR, however, the repair performance of a single fix pattern is not studied. We revisit the performance of template-based APR to build comprehensive knowledge about the effectiveness of fix patterns, and to highlight the importance of complementary steps such as fault localization or donor code retrieval. To that end, we first investigate the literature to collect, summarize and label recurrently-used fix patterns. Based on the investigation, we build TBar, a straightforward APR tool that systematically attempts to apply these fix patterns to program bugs. We thoroughly evaluate TBar on the Defects4J benchmark. In particular, we assess the actual qualitative and quantitative diversity of fix patterns, as well as their effectiveness in yielding plausible or correct patches.
- *Debugging Method Names*: Except the issues about semantic/static bugs in programs, we note that how to debug inconsistent method names automatically is important to improve program quality. In this dissertation, we propose a deep learning based approach to spotting and refactoring inconsistent method names in programs.

To my dear wife Lie Ma.

Acknowledgements

This dissertation would not have been possible without the support of many people who in one way or another have contributed and extended their precious knowledge and experience in my PhD studies. It is my pleasure to express my gratitude to them.

First of all, I would like to express my deepest thanks to my supervisor, Prof. Yves Le Traon, who has given me this great opportunity to come across continents to pursue my doctoral degree. He has always trusted and supported me with his great kindness throughout my whole PhD journey.

Second, I am equally grateful to my daily advisers, Dr. Tegawendé Bissyandé and Dr. Dongsun Kim, who have introduced me into the world of automated program repair. Since then, working in this field is just joyful for me. I am particularly grateful to them as well as Dr. Jacques Klein, for their patience, valuable advice and flawless guidance. They have taught me how to perform research, how to write technical papers, and how to conduct fascinating presentations. Their dedicated guidance has made my PhD journey a fruitful and fulfilling experience. I am very happy for the friendship we have built up during the years.

Third, I would like to extend my thanks to all my co-authors including Prof. Sin Yoo, Prof. Martin Monperrus, Prof. Suntae Kim, Dr. Li Li, Anil Koyuncu, Kisub Kim, Pingfan Kong, Taeyoung Kim, etc. for their valuable discussions and collaborations.

I would like to thank all the members of my PhD defense committee, including chairman Dr. Tegawendé Bissyandé, Prof. David Lo, Prof. Andreas Zeller, my supervisor Prof. Yves Le Traon, and my daily adviser Dr. Dongsun Kim. It is my great honor to have them in my defense committee and I appreciate very much their efforts to examine my dissertation and evaluate my PhD work.

I would like to also express my great thanks to all the friends that I have made in the Grand Duchy of Luxembourg for the memorable moments that we have had. More specifically, I would like to thank all the team members of SerVal at SnT for the great coffee breaks and interesting discussions.

Finally, and more personally, I would like to express my deepest appreciation to my dear wife Lie Ma, for her everlasting support, encouragement, and love. Also, I can never overemphasize the importance of my mother in shaping who I am and giving me the gift of education. The last but not least, I would like to extend my thanks to my daughter Ruoshui Liu who gives me unexpected happiness and motivation for this dissertation. Without their support, this dissertation would not have been possible.

Kui Liu
University of Luxembourg
December 2019

Contents

List of figures

List of tables

1 Introduction

In this chapter, we first introduce the motivation of automated program repair. Then, we summarize the challenges people face when conducting automated program repair, and finally we present the contributions and roadmap of this dissertation.

Contents

1.1 Motivation

As software is pervasive in numerous domains in the real world, the reliability of software programs is of critical importance since issue-impacted programs may jeopardize the digital assets of software usages [235]. Thus, software quality is primarily concerned by practitioners. However, software quality could always be adversely impacted by various problems, especially program bugs (e.g., semantic bugs [141] and static bugs [90]). On the other hand, code identifiers (e.g., method names) with low quality, that cannot present program behaviors clearly, could impact the readability and maintainability of programs, and even lead to bugs [2, 191]. In practice, mature software projects could be shipped with known or unknown bugs into the market [132] since there is little resource (measured in time, money, or people) for the current software projects available to address them [16]. For example, Microsoft shipped and released Windows 2000 with more than 63,000 bugs, which are already known and filed by its developers [146].

These bugs could always be encountered by software users, which could further lead to huge financial losses. For example, Amazon's site went down in 2016 for 20 minutes, it's estimated that the outage cost Amazon around $3.75 million. As reported by National Institute of Standards and Technology in 2002, software bugs cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product[1]. In 2016, this number jumped to $1.1 trillion[2]. Furthermore, Zhivich and Cunningham [253] reported that the system for radiation treatment in a hospital incorrectly calculated radiation dosages, which resulted in several patients receiving deadly radiation overdoses during their treatment that led to at least eight patients died.

To reduce the huge losses caused by program bugs, developers should address them in time. However, manually fixing bugs is a time-consuming, expensive and difficult task. As stated in a technical report from the University of Cambridge, software developers have to spend 50% of their time on fixing bugs [31]. Furthermore, some bugs could be existed for a long time until they are addressed. For example, Kim and Whitehead [107] reported that the median time for fixing a single bug is about 200 days. Koyuncu et al. [111] also reported that generating DLH (Detection and Localization are automated but Human written) patches for bugs in Linux Kernel will cost as much as a median time of 260 days.

Except the excessive time consumption for fixing bugs, it spends exceeded cost of other resources as well. The technical report from Cambridge University also has shown that the annual global cost of general debugging is 312 billion dollars [31]. A study showed that a typical software project will spend 90% of its total cost on debugging after its delivery [200]. Additionally, mature software projects are continuously released with bugs day by day. For example, in 2005, one Mozilla developer claimed that, "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle." [17, p. 363]. The number of outstanding software defects typically exceeds the resources available to address them [16] .

Manually fixing program bugs is a difficult and excessively high-costing process, which is a suffocative burden for developers. The expensive cost of debugging makes developers unable to focus on other creative tasks such as software design. Rather, most developers tend to spend their time repairing bugs, which makes software development less productive. To reduce the excessive cost and the developer's burden on fixing bugs, automated program repair techniques must be devised.

Automated program repair (APR) holds the promise of reducing manual debugging effort by automatically generating patches for defects identified in a program. In production, APR will drastically reduce time-to-fix delays and limit downtime. In a development cycle, APR can help suggest changes to accelerate debugging. In the literature, indeed, APR has shown to be an effective approach to automatically fixing bugs. Nevertheless, many problems on program repair are still open to be explored.

[1] http://www.abeacha.com/NIST_press_release_bugs_cost.htm
[2] https://medium.com/@ryancohane/financial-cost-of-software-bugs-51b4d193f107

In this dissertation, we explore program repair approaches from two scenarios: 1) test-suite-based automated program repair, which relies on the failed tests to localize bugs and validate patches; and 2) debugging method names, which aims at spotting and refactoring the inconsistent method names (i.e., the poor method names that cannot present the program behavior clearly and correctly) in programs.

1.2 Challenges of Program Repair

In this section, we present the challenges of test-suite-based automated program repair and debugging method names. We first summarize the challenges for test-suite-based automated program repair. The basic automated program repair pipeline consists of three processes: fault localization, patch generation and patch validation, as shown in Figure 1.1. We introduce the technical challenges in the repair pipeline we face when performing automated program repair on real bugs. More specifically, we introduce the challenges that are specific to fault localization used in automated program repair and that are inherited from the patch generation of automated program repair approaches, respectively. On the other hand, we summarize the challenges on debugging method names as well.

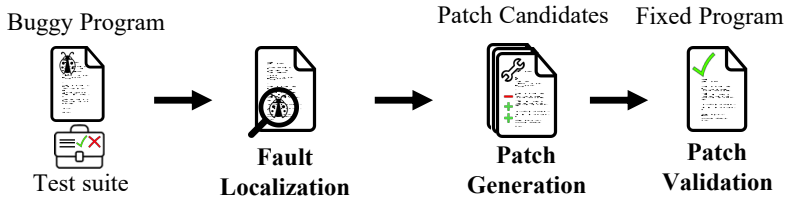


Figure 1.1: The Basic Pipeline of Automated Program Repair.

1.2.1 Challenges in Fault Localization

Once a fault arises, most recent APR systems follow the same basic pipeline as shown in Figure 1.1 to start with the fault localization (FL). The FL step identifies a code entity in a buggy program as the potential fault location which should be addressed. We now present some challenges for automated program repair that are mainly due to the fault localization techniques.

Spectrum based fault localization (also referred to as coverage-based fault localization) [234, 241, 252] is the widely used FL technique in APR community [43, 47, 102, 104, 122–124, 128, 141, 143, 147, 148, 150, 164, 179, 224, 237, 239, 240], which relies on the execution of test cases to track the detailed execution information of a program from certain perspectives, such as execution information for statements, conditional branches, loop-free intra-procedural paths or methods [80]. Nevertheless, the automated fault localization techniques are not mature and still under being explored with open issues [187, 234]. For example, GZoltar [40] is a framework for automating the testing and debugging phases of the software development life-cycle, which has been widely used in the automated program repair systems for Java programs [93, 141–143, 158, 226, 237, 239]. However, its early version (i.e., GZoltar-v0.1) cannot localize the bugs located in constructor methods since it failed to execute the test cases for constructors. This issue was addressed in a later version (i.e., GZoltari-v1.6), but it has arisen again in its latest version (i.e., GZoltar-v1.7.2). If the FL techniques cannot find the bug, it is impossible for APR systems to fix it [140], which is a primary challenge for automated program repair.

Spectrum based FL techniques rely on statement execution frequencies to localize the bug position; the more frequent a statement is executed by failing tests and the less often it is executed by passing tests, the more suspicious the statement is to be faulty [187]. In the literature, such suspiciousness is calculated by different ranking metrics, such as Tarantula [97], Ochiai [4], Op2 [177], Barinel [5] and

DStar [232]. These ranking metrics compute the suspiciousness score of a given source code statement (s) with the formulas as below:

$$Suspiciousness(s) = \frac{\frac{failed(s)}{failed(s)+failed(\neg s)}}{\frac{failed(s)}{failed(s)+failed(\neg s)} + \frac{passed(s)}{passed(s)+passed(\neg s)}} \quad (Tarantula) \quad (1.1)$$

$$Suspiciousness(s) = \frac{failed(s)}{\sqrt{(failed(s) + passed(s)) * (failed(s) + failed(\neg s)}} \quad (Ochiai) \quad (1.2)$$

$$Suspiciousness(s) = failed(s) - \frac{passed(s)}{passed(s) + passed(\neg s) + 1} \quad (Op2) \quad (1.3)$$

$$Suspiciousness(s) = 1 - \frac{passed(s)}{failed(s) + passed(s)} \quad (Barinel) \quad (1.4)$$

$$Suspiciousness(s) = \frac{failed(s)^*}{passed(s) + failed(\neg s)} \quad (DStar) \quad (1.5)$$

where $failed(s)$ and $passed(s)$ respectively denote the number of failed and passed tests that executed statement s, while $failed(\neg s)$ and $passed(\neg s)$ respectively present the number of failed and passed tests that do not execute statement s. Eventually, with a related ranking metric, FL techniques report a ranked list of statements associated with the suspiciousness scores. Obviously, the ranked list would vary as the selected ranking metric. Therefore, it is the other challenge of fault localization for boosting automated program repair since it is impossible to declare which ranking metric is the best for automated program repair.

In the APR community, APR systems generally focus on the patch generation step, but tend to use similar strategies for fault localization. Their bug fixing performance is measured by counting the number of bugs for which the system can generate a patch that passes all test cases. Therefore, it would be biased to compare their repair performance when their fault localization is not set on the same baseline [140]. Nevertheless, given the growing interest in APR among software engineers, it is important to ensure that the research outputs are relevant and well-assessed in terms of reliable performance for practitioners.

If the fault localization configurations are not set on the same baseline, it would be difficult to assess whether the repair performance is improved by the advanced fault localization technique or the advanced patch generation approach. To boost the automated program repair, it should fairly clarify the contributions of repair performance from patch generation approaches or fault localization techniques. Otherwise, it would bias the evaluation results, or even mislead the participators to contribute to the APR community. To sum up, the bias from fault localization to be the other challenge for automated program repair.

1.2.2 APR-Inherited Challenges

Once a bug is localized, APR systems will generate patch candidates for it (c.k., patch generation in Figure 1.1), which consists of the key contribution of APR. To date, the state-of-the-art APR systems have achieved promising results on fixing real bugs in a well-established benchmark (i.e., Defects4J [98]). When counting the number of bugs that are fixed with plausible patches (i.e., they make the programs pass all the test cases) by the state-of-the-art is not high enough comparing with the total number of bugs in the benchmark. If only considering the bugs are fixed with correct patches (i.e., they are equivalent to the patches that were actually submitted by the program developers), the number is much smaller. Furthermore, some APR systems (e.g., APR approaches based on genetic programming) could generate a large number of nonsensical patches [104]. Improving the efficiency of APR in terms of quantity and quality is the primary APR-inherited challenge.

In the APR community, fix templates (also referred to as fix patterns [144]) are widely used in various APR systems to generate patch candidates [104, 147]. However, those fix templates are manually

summarized from human-written patches or pre-defined by researchers. Manual summarization of fix patterns is a heavy burden for APR practitioners. In addition, manual mining of fix patterns cannot enumerate the common and effective fix patterns as many as possbile. Long et al. [147] proposed Genesis to infer fig patterns from real-world patches, but it only focuses on three kinds of bugs. Therefore, automatically mining fix patterns is the other challenge contributed to automated program repair.

1.2.3 Challenges of Debugging Method Names

Similar to the basic pipeline of automated program repair as shown in Figure 1.1, debugging method names also consists of three basic process: localization, generation and validation. Method names summarize the intuitive and vital information for developers to understand the behavior of programs or APIs [23, 53, 54, 67]. They are descriptive natural language but not constructed logic program code. Thus, there is no any test suite that can be used to spot the inconsistent method names, generate and validate the new generated method name candidates. The intrinsic characteristic of method names hardens the process of localizing the inconsistent method names as well as generating and validating the new generated method name candidates. Fortunately, knowledge engineering, especially the nature language processing, makes it is possible to build the semantic connection between method names and method body code, which could be leveraged to debug method names. In this dissertation, we propose a deep learning based approach to debugging method names.

1.3 Contributions

We now summarize the contributions of this dissertation as below:

- **A closer look at real-world patches.** We investigate the real-world patches in a fine-grained way to deepen the knowledge of real-world patches for APR, via a systematic study of 16,450 bug fix commits from seven Java open-source projects. We find that there are opportunities for APR techniques to improve their effectiveness by looking at code elements that have not yet been investigated. We also discuss nine insights into tuning automated repair tools. For example, a small number of statement and expression types are recurrently impacted by real-world patches, and expression-level granularity could reduce the search space of finding fix ingredients, where previous studies never explored.
  This work has led to a research paper published to the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018).
- **An investigation of fault localization bias in benchmarking automated program repair systems.** We provide a clear view in the fault localization (FL) settings of the state-of-the-art APR systems and investigate the impact of FL setting on the repair performance of APR systems. Specifically, we identify and investigate a practical bias caused by the FL in a repair pipeline. We propose to highlight the different FL configurations used in the literature, and their impact on APR systems when applied to the Defects4J benchmark. Then, we explore the performance variations that can be achieved by "tweaking" the FL step.
  This work has led to a research paper published to the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST 2019).
- **A deep learning based fix pattern mining for static analysis bugs.** We provide an overall view on the distributions of the bugs spotted by a static analysis technique (i.e., FindBugs [90]) and propose a deep learning based approach to mining fix patterns for these bugs. We first collect and track a large number of fixed and unfixed violations across revisions of software. The empirical analyses reveal that there are discrepancies in the distributions of violations that are detected and those that are fixed, in terms of occurrences, spread and categories, which can provide insights into prioritizing violations. To automatically identify

patterns in violations and their fixes, we propose an approach that utilizes convolutional neural networks to learn features and clustering to regroup similar instances and mine common fix patterns for them. Our evaluation shows that the mined fix patterns could be used to fix similar static analysis bugs and some semantic bugs.
This work has led to a research paper accepted by the IEEE Transactions on Software Engineering in 2018.

- AVATAR **: An automated program repair system based on fix patterns of static analysis violations.** We propose to fix semantic bugs with the fix patterns of static analysis valitions. Fix pattern-based patch generation is a promising direction in the APR community. Notably, it has been demonstrated to produce more acceptable and correct patches than the patches obtained with mutation operators through genetic programming. The performance of pattern-based APR systems, however, depends on the fix ingredients mined from fix changes in development histories. Unfortunately, collecting a reliable set of bug fixes in repositories can be challenging. We propose to investigate the possibility in an APR scenario of leveraging code changes that address violations by static bug detection tools. To that end, we build the AVATAR APR system, which exploits fix patterns of static analysis violations as ingredients for patch generation. Evaluated on the Defects4J benchmark, we show that, the two fix pattern based APR systems can achieve promising repair performance that is comparable to the state-of-the-art approaches in the literature.
This work has led to a research papers published to the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2019).
- **TBar: A fix template based automated program repair system.** We revisit the performance of template-based APR to build comprehensive knowledge about the effectiveness of fix patterns, and to highlight the importance of complementary steps such as fault localization or donor code retrieval. To that end, we first investigate the literature to collect, summarize and label recurrently-used fix patterns. Based on the investigation, we build TBar, a straightforward APR tool that systematically attempts to apply these fix patterns to program bugs. We thoroughly evaluate TBar on the Defects4J benchmark. In particular, we assess the actual qualitative and quantitative diversity of fix patterns, as well as their effectiveness in yielding plausible or correct patches.
This work has led to a research paper published to the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019).
- **A deep learning based approach to debugging method names.** We propose to debug inconsistent method names with deep learning frameworks. To ensure code readability and facilitate software maintenance, program methods must be named properly. In particular, method names must be consistent with the corresponding method implementations. Debugging method names remains an important topic in the literature, where various approaches analyze commonalities among method names in a large dataset to detect inconsistent method names and suggest better ones. We note that the state-of-the-art does not analyze the implemented code itself to assess consistency. We thus propose a novel automated approach to debugging method names based on the analysis of consistency between method names and method code. The approach leverages deep feature representation techniques adapted to the nature of each artifact. Experimental and live study results show that our proposed approach can effectively debug inconsistent method names in the real-world code bases.
This work has led to a research paper published to the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019).

1.4 Roadmap

Figure 1.2 illustrates the roadmap of this dissertation. Chapter 2 gives a brief introduction on the necessary background information, including automated program repair, fix pattern mining and

debugging method names. In Chapters 3, we present an empirical study on real-world patches in a fine-grained way to deepen the knowledge on patches for APR. In Chapter 4, we present the distribution of fault localization usages in the APR community and investigate the fault localization bias in benchmarking automated program repair. In Chapters 5, we present a deep learning based approach to mining fix patterns for FindBugs violations. In Chapters 6, we present an APR system with the fix patterns for static analysis violations. In Chapters 7, we implement a fix pattern based APR system, TBar, to revisit the repair performance of fix patterns released in the literature. In Chapters 8, we propose a deep learning based approach to spotting and refactoring inconsistent method names. Finally, in Chapter 9, we conclude this dissertation and discuss some potential future works.

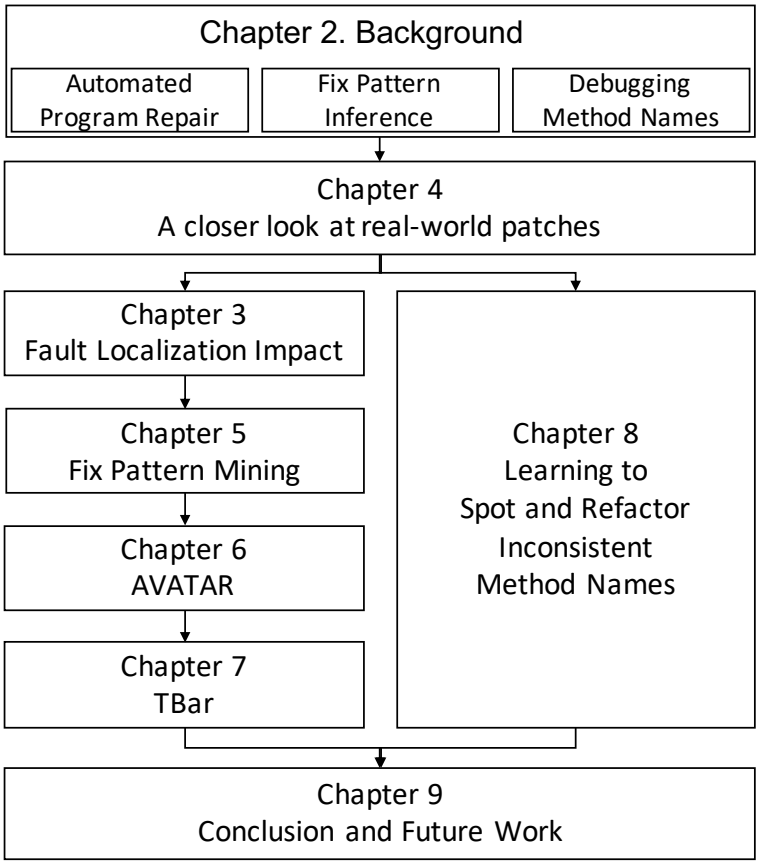


Figure 1.2: Roadmap of This Dissertation.

2 Background

In this chapter, we provide the preliminary details that are necessary to understand the purpose, techniques and key concerns of the various research studies that we have conducted in this dissertation. Mainly, we revisit some details of automated program repair, fix pattern mining and debugging method names, respectively.

Contents

2.1 Automated Program Repair

Automated program repair (APR) holds the promise of reducing manual debugging effort by automatically generating patches for defects identified in a program. In production, APR will drastically reduce time-to-fix delays and limit downtime. In maintenance, APR can help suggest changes to accelerate debugging. There are two distinct repair scenarios in the literature: (1) fixing syntactic errors, i.e., cases where code violates some programming language specifications [27,75] and (2) fixing *semantic bugs*, i.e., cases where implementation of program behaviour deviates from a developer's intention [163,179]. The latter requires the failed test case(s) for the bug localization and patch validation, which is the scope of automated program repair explored in this dissertation.

2.1.1 State-of-the-Art APR

Since the milestone work of Weimer et al. [224] ten years ago, APR has progressively become an essential research field. Various APR approaches [43,47,91,93,102,104,122–124,128,138,140,141,147,148,150,164,179,224,226,239,240] have been proposed in the literature, which are summarized into three categories: heuristic-based repair, constraint-based repair, and learning-based repair [129].

Heuristic-based repair approaches employ a generate-and-validate methodology, which constructs and iterates over a search space of syntactic program modifications [129]. The search space denotes a set of considered modifications (also referred to as patch candidates [104]) for a buggy program that are generated by a dedicated repair approach. The validate proceeds to count the number of tests that pass when a patch candidate has been applied to a buggy program. If a patch candidate can make a buggy program pass all tests, it is considered as the patch for the buggy program. Associated APR tools for Java programs include jGenProg [158], GenProg-A [249], ARJA [249], RSRepair-A [249], SimFix [93], jKali [158], Kali-A [249], jMutRepair [158], HDRepair [124], PAR [104], S3 [122], ELIXIR [199], SOFix [144], CapGen [226], and so on.

Constraint-based repair approaches proceed a methodology different from heuristic-based repair, which constructs repair constraints that will be used to select donor code for the patch generation [75]. The constraints describe the code fragments that should satisfy the variable types, values or behaviour specified by the constraints. Such code fragments are returned as the potential matches, which further be synthesized into patch candidates with the specific functions. For example, symbolic execution approaches extract properties about the function to be synthesized; these properties constitute the repair constraints. Solutions to the repair constraints can be obtained by constraint solving or other search techniques. Nopol [240], DynaMoth [57] ACS [239], and Cardumen [159] are constraint-based repair tools that are dedicated to repairing buggy `if` conditions and to add missing `if` preconditions with specific constraints.

Learning-based repair approaches explore the advanced machine learning technique, especially deep learning technique, to boosting program repair [129]. The availability of large numbers of human-written patches from real world enable the study of learning-based repair [75]. To date, learning-based repair has been exploited in three ways: 1) learning models of correct code that are used to prioritize the patch candidates in terms of the correctness [150]; 2) learning code transformation patterns that summarize how human-written patches change buggy code into correct code [32,147], where patterns can be further used to generate patch candidates; and 3) learning to improve the repair process and training models for end-to-end repair [147,219], where models are leveraged to predict the correct code for the given buggy code without using any other explicitly provided context information [129].

In this dissertation, we explore fix pattern based program repair approaches, which could be grouped into heuristic-based repair as the main process of fix pattern based repair is consistent with heuristic-based repair. A fix pattern consists of a code entity (e.g., a statement and an expression) and a related repair action (e.g, update and insert) [137]. To repair bugs with fix patterns, it needs to select

appropriate fix patterns for bugs, which is conducted by matching the AST context information of buggy code with the code entity of fix patterns. The repair action guides to modify the code entity at Abstract Syntax Tree (AST) level to generate patch candidates by transforming the buggy code into potential correct code with appropriate donor code selected from the buggy program with dedicated searching strategies. The validation of fix pattern based repair is the same as the validation process of heuristic-based repair with the regression test.

2.1.2 Fault Localizaiton in APR

As shown in Figure 1.1, the basic repair pipeline starts with the fault localization that identifies the suspicious bug positions which should be addressed by APR systems. In the literature, APR systems generally focus on the patch generation step, but tend to use similar strategies for fault localization. The systems often rely on a testing framework such as GZoltar [40], and a spectrum-based fault localization formula [234, 241, 252], such as Ochiai [3].

In the APR community, most APR systems for Java Programs use the same testing framework (i.e., GZoltar [40]) and the same spectrum-based fault localization formula (i.e., Ochiai [3]). For example, SimFix [94] uses a later version of the GZoltar (i.e, GZoltar-v1.6.2). Nopol [240] uses an even older version, i.e., GZoltar-v0.0.1, while others [113, 141–143, 158, 226, 237, 239] use other GZoltar-v0.1.1. There are some APR systems [43, 91, 124, 199] that leverage other testing framework to localization bug positions.

On the other hand, the spectrum-based fault localization relies on a formula calculating the suspiciousness of program elements (e.g., Ochiai) to spot the bug positions. In the APR community, the popularity of Ochiai is backed up by empirical evidence on its effectiveness to help localize faults in object-oriented programs as highlighted by several fault localization studies [186, 208, 236, 241]. A recent work by Pearson et al. [187] has even shown that Ochiai outperforms current state-of-the-art ranking metrics, or at least offers similar performance measures.

In the patch generation process, APR systems take input a ranked list of suspicious positions spotted by the fault localization process. Such lists are eventually decided by the testing framework and the suspiciousness calculating formula of fault localization techniques. If the lists are not calculated based on the same framework or formula, the input for the patch generation of APR systems will not be on the baseline, which could bias the evaluation of repair performance for APR systems.

In the APR community, some APR systems leverage supplementary information to assist the fault localization step and improve accuracy. For example, HDRepair [124], JAID [43] and SketchFix [91] assume that the faulty methods are known: the fault localization step therefore focuses on ranking the lines inside the method, thus leaving out noisy statements that other APR tools are considering. This artificially reduces the probability to produce overfitting patches for these tools, and even increases the chance to generate a correct patch before any execution timeout. ssFix [237] prioritizes statements from the stack trace of crashed programs that are executed before those statements that are ranked by the FL tool. ACS [239] uses predicate switching [251] and refines the suspicious code locations list. SimFix [93] applies a test case purification approach to improve the accuracy of fault localization step before patch generation. Although these extra steps, which are taken to supplement fault localization step, could be justified intuitively, the community needs to clearly investigate their impact, in order to enable fair comparisons among the repair techniques themselves.

In this dissertation, we propose to investigate the fault localization impact on the repair performance of APR systems. Indeed, given that APR systems are currently compared with respect to the number of bugs that are correctly fixed, it is important that the research community reflects on what are the key contributions for explaining APR performance: for example, by counting numbers of correct patches, several programs may not be repairable by a given APR system simply because the fault is not accurately localized by the implemented fault localization step.

2.1.3 APR Assessment

The assessment of APR approaches in the literature attempts to provide information on the number of bugs for which APR tool can generate a patch that makes the buggy program pass all the test cases [93, 104, 124, 158, 199, 224, 240]. Six years after this seminal work on generate-and-validate patch generation systems, Qi et al. [194] and Smith et al. [202] have concurrently presented empirical results showing that generated patches can be plausible only: they can make the programs pass all the test cases but may not actually fix the bug, due to overfitting on the test suite. Therefore, researchers have started to focus some effort in automating the identification of patch correctness [238]. Eventually, to fairly assess the performance on fixing real bugs of APR tools, the number of bugs for which a correct (i.e., it is semantically equivalent to the patch that the program developer accepts for fixing the bug) patch is generated appeared to be a more reasonable metric than the mere number of plausible patches [239]. This metric has since then become standard among researchers, and is now widely accepted in the literature for evaluating APR tools [43, 91, 93, 141–144, 199, 226]. Based on data presented with such metric, researchers explicitly rank the APR systems, and use this ranking as a validation of new achievements in program repair. In this dissertation, we follow such metric to proceed the evaluation of our proposed APR systems.

2.2 Fix Pattern Inference

An early strategy of APR is to generate concrete patches based on fix patterns [104] (also referred to as fix templates [144] or program transformation schemas [91]). This strategy is now common in the literature and has been implemented in several APR systems [56, 91, 104, 113, 140, 141, 144, 159, 199]. To that end, fix patterns have to be collected before implementing the related APR systems. In the literature, fix patterns have been mainly explored with the four ways: manual summarization, pre-definition, statistics and mining.

1. **Manual Summarization**: Pan et al. [185] manually identified 27 fix patterns from patches of five Java projects to characterize the fix ingredients of patches. Kim et al. [104] manually summarized 10 fix patterns from 62,656 human-written patches collected from Eclipse JDT.
2. **Pre-definition**: Durieux et al. [56] pre-defined 9 repair actions for null pointer exceptions by unifying the related fix patterns proposed in previous studies [55, 103, 151]. On the top of *PAR* [104], Saha et al. [199] further defined 3 new fix patterns to improve the repair performance. Hua et al. [91] proposed an APR tool with six pre-defined so-called code transformation schemas. Xin and Reiss [237] proposed an approach to fixing bugs with 34 predefined code change rules at the AST level.
3. **Statistics**: Besides formatted fix patterns, researchers [93, 226] also explored to automate program repair with code change instructions (at the abstract syntax tree level) that are statistically recurrent in existing patches [93, 139, 157, 225, 254]. The strategy is then to select the top-n most frequent code change instructions as fix ingredients to synthesize patches.
4. **Mining**: Long et al. [147] proposed Genesis, to infer fix patterns for three kinds of defects from existing patches. Liu and Zhong [144] explored fix patterns from Q&A posts in Stack Overflow. Koyuncu et al. [113] mined fix patterns at the AST level from patches by using code change differentiating tool [60]. In general, mining approaches yield a large number of fix patterns, which are always about addressing deviations in program behavior.

In this dissertation, we propose a deep learning based approach to mining fix patterns for static analysis bugs.

2.3 Debugging Method Names

Method names in programs are the intuitive and vital information for developers to understand the behavior of programs or APIs [23, 53, 54, 67]. If a method has a good name, developers do not need to

look at the method body [62]. The inconsistent method names, that cannot present the program behavior correctly, can make programs harder to understand and maintain [19,20,87,119,133,212,228], and may even lead to software defects [1,2,15,36,191].

To debug method names, Høst and Østvold [89] explored method naming rules and semantic profiles of method implementations. Kim *et al.* [108] relied on a custom code dictionary to detect inconsistent names. Allamanis *et al.* introduced the NATURALIZE framework [9] learning the domain-specific naming convention from local contexts to improve the stylistic consistency of code identifiers with n-gram model [33]. Then, building on this framework, they proposed a log-bilinear neural probabilistic language model to suggest method and class names with similar contexts [10]. The researchers leveraged attentional neural networks [12] to extract local time-invariant and long-range topical attention features in a context-dependent way to suggest names for methods. More recently, Pradel and Sen [191] proposed a learning approach to detect buggy names, which reasons about names based on a semantic representation and which automatically learns bug detectors instead of manually writing them. In this dissertation, we also propose a deep learning approach to learning features for method names and method bodies, respectively. We further leverage the learned features to spot and refactor inconsistent method names in programs.

3 A Closer Look at Real-World Patches

In this empirical study, we contribute to building the knowledge on repair actions via a systematic and fine-grained study of 16,450 bug fix commits from seven Java open-source projects. We find that there are opportunities for APR techniques to improve their effectiveness by looking at code elements that have not yet been investigated. We also discuss nine insights into tuning automated repair tools, challenges and possible resolves. For example, a small number of statement and expression types are recurrently impacted by real-world patches, and expression-level granularity could reduce search space of finding fix ingredients, where previous studies never explored.

This chapter is based on the work published in the following research paper:

- Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. A closer look at real-world patches. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, pages 275–286. IEEE, 2018

Contents

3.1 Overview

In recent years, to reduce the cost of software bugs [181], the research community has invested substantial efforts into automated program repair (APR) approaches [22, 42, 43, 47, 102, 122–124, 128, 147, 148, 157, 164, 240]. The first significant milestone in that direction is GenProg [224], an APR technique that uses genetic programming to apply a sequence of edits on a buggy source code until a test suite is satisfied. After GenProg, several follow-up techniques have been proposed: Nguyen et al. [179] relied on symbolic execution. Xiong et al. [239] focused on mining contextual information from documents and across projects. Kim et al. [104] proposed to leverage fix templates manually mined from human-written patches. Long and Rinard [150] built a systematic approach to leveraging past patches.

Although existing APR studies have achieved promising results, two issues mainly stand out: the applicability of APR techniques for a diverse set of bugs, and the low quality of patches generated by them [173, 254]. The existing APR techniques tend to generate patches for few specific types of bugs [104] or to make nonsensical changes to programs [128].

In this study, our conjecture is that one potential issue with the (in)effectiveness of APR techniques is the limitation carried by the granularity at which APR techniques perform repair actions. Since *generate-and-validate* [174] approaches (such as GenProg) rely on fault localization techniques to identify buggy code, they generate patches at the statement level, often based on stochastic mutations. Actually, as our study shows, most bugs are localized on specific code entities (e.g., the wrong `Infix-expression` in Figure 3.1) within buggy statements. Therefore, mutating every part of a buggy statement is likely to lead, at best, to a very slow and resource-intensive fixing process, and at worst, to the generation of incorrect and nonsensical patches with high costs.

Real-world patches (i.e., written by human developers) can provide useful information (e.g., on repair actions) for efficient generation of correct patches. Previous work [104, 124, 147, 150] has already shown that patches from software repositories can be leveraged to improve software repair. Nevertheless, the prerequisite for further advancing state-of-the-art APR techniques is to acquire all-round and detailed understanding about real-world patches.

Several studies in the literature have attempted to build such knowledge, but all focused on characterizing changes at the *statement* level. Pan et al. [185] manually summarized 27 fix patterns from existing patches of Java projects. Their patterns are, however, in a high-level form (e.g. "If-related: Addition of Post-condition Check (`IF-APTC`)"). Martinez et al. [157] analyzed bug fix transactions at the AST statement level of code changes. In a concurrent study, Zhong et al. [254] also analyzed the repair actions of patches at the AST statement level to understand the nature of bugs and patches (see Section 3.7 for more detailed comparison). Although these studies provide interesting insights into bug fix patterns at the coarse-grained level of statements, they can be misleading for implementing automated repair actions. Indeed, buggy parts can be localized in a more fine-grained way, leading to more accurate repair actions.

Consider the real-world code change illustrated in GNU Diff format in Figure 3.1. This change is committed to a software repository as a *patch*.

Most fault localization tools would identify Line 183 in file *MultivariateNormalDistribution.java* as a suspicious (i.e., might be buggy) location before commit `cedf0d` of project `commons-math`. To generate a corresponding patch, APR tools will apply generic fix patterns for `ReturnStatement`, thus it may fail to properly fix the bug or take a long time even if it succeeds.

Based on the hierarchical and fine-grained view (see Figure 3.2) of repair actions of the patch in Figure 3.1 provided by GumTree (an AST based code changes differencing tool) [60], it is easy to find that the exact repair action of this patch occurs to "`-dim / 2`", an `InfixExpresion` node in the AST, a child of a `MethodInvocation` node (see Figure 3.3). This repair action is more precise than replacing a `ReturnStatement` as a whole with another statement. Our conjecture is that it is

```
1 Commit cedf0d27f9e9341a9e9fa8a192735a0c2e11be40
2 --- src/main/java/org/apache/commons/math3/distribution/MultivariateNormalDist
    ribution.java
3 +++ src/main/java/org/apache/commons/math3/distribution/MultivariateNormalDist
    ribution.java
4 @@ -183, 3 +183, 3 @@
5 -   return FastMath.pow(2 * FastMath.PI, -dim / 2) *
6 +   return FastMath.pow(2 * FastMath.PI, -0.5 * dim) *
7           FastMath.pow(covarianceMatrixDeterminant, -0.5) *
8           getExponentTerm(vals);
```

Figure 3.1: Patch of fixing bug MATH-929, a value-truncated bug.

The hierarchical repair actions of a patch parsed by GumTree:
UPD ReturnStatement@@"***buggy code***" to "***fixed code*** ".
UPD InfixExpression@@"***buggy code***" to "***fixed code***".
UPD MethodInvocation@@"***buggy code***" to "***fixed code***".
UPD InfixExpression@@"-dim / 2" to "-0.5 * dim".
UPD PrefixExpression@@ "-dim" to "-0.5".
DEL SimpleName@@ "dim" from "-dim" .
INS NumberLiteral@@"0.5" to "-dim".
UPD Operator@@"/" to "*".
DEL NumberLiteral@@ "2" from "2".
INS SimpleName@@"dim" to "2".

Figure 3.2: A graphic representation of the hierarchical repair actions of a patch parsed by GumTree. Buggy code entities are marked with red and fixed code entities are marked with green. 'UPD' represents updating the buggy code with fixed code, 'DEL' represents deleting the buggy code, 'INS' represents inserting the missed code, and 'MOV' represents moving the buggy code to a correct position. Due to space limitation, the buggy and fixed code (See Figure 3.1) are not presented in this figure.

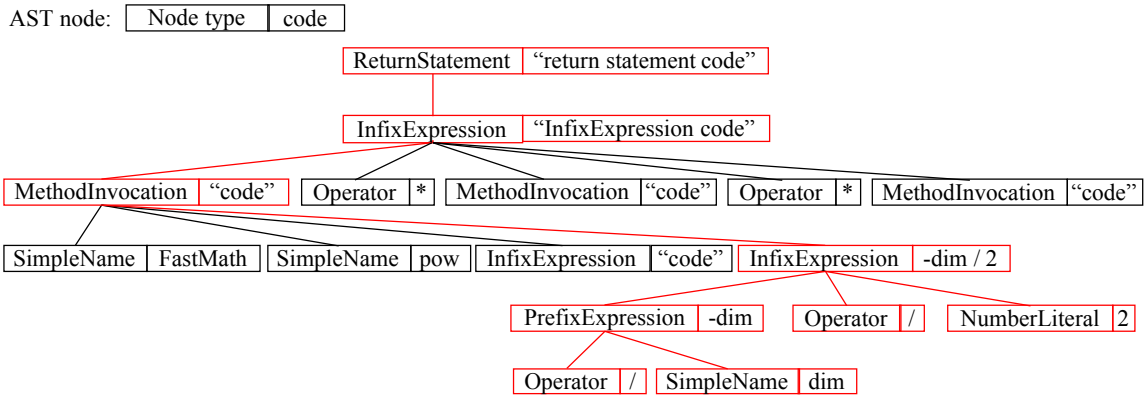


Figure 3.3: Example of parsing buggy code in terms of AST. The exact buggy code entities in its AST are highlighted with red. For simplicity and readability of the AST of buggy code, the sub-trees of bug-free code nodes are not shown in this tree.

less probable to find or identify exactly the same repair action with statement replacement than expression replacement, from the search space of human-written patches.

As shown in the above motivating example, previously existing studies [157, 185, 254] did not take a close look at existing patches using advanced tools such as GumTree and APR research may have been missing important and accurate insights for enhancing the state-of-the-art APR techniques. In particular, by mining patches beyond statement-level information, we can investigate which fine-

grained buggy code entities are recurrent in repair changes, and which repair actions are successfully applied to them. Insights from such questions can allow tuning APR techniques towards faster completions (e.g., focus changes on more likely buggy entities) and more accurate generation of correct patches (e.g., make accurate changes). To that end, we investigate 16,450 bug fix commits collected in two distinct ways from seven Java open source project repositories, in a fine-grained way by leveraging GumTree.

Looking closely at real-world patches, we find that there are opportunities for APR techniques to be targeted at code elements that have not yet been investigated. We also find expression-level granularity could reduce search space of finding fix ingredients for similar bugs. We further discuss nine insights into tuning APR techniques towards being fast in their trials for patch generations, but also towards producing patches that have more probability to be correct.

3.2 Background

This section clarifies the notions of code entities related to AST representations and AST diffs of code changes.

3.2.1 Code Entities

Code entities are basic units (i.e., nodes) comprising ASTs. Since our work investigates Java projects, this study collects code entities defined in the Eclipse JDT APIs[1]. The APIs describe code entities in the AST representation of Java source code. There are 22 statement[2] (e.g., `ReturnStatement`), declarations (e.g., `TypeDeclaration`, `EnumDeclaration`, `MethodDeclaration` and `FieldDeclaration`), and 35 expression[3] (e.g., `InfixExpression`) entity types. We collect these code entities from Java source code. Note that we refer direct children nodes of a statement or an expression in an AST as *code element*s in this study.

3.2.2 AST Diffs

Our study analyzes patches in the form of AST diffs. In contrast with GNU diffs that represent code changes as a pure text-based line-by-line edit script, AST diffs provide a hierarchical representation of the changes applied to the different code entities at different levels (statements, expressions, and elements). We leverage GumTree [60] to extract and describe repair actions implemented in patches since the tool is open source[4], allowing for replication, and is built on top of the Eclipse Java model[5].

Overall, in this study:

- A *Code entity* represents a *node* in ASTs. It can be a declaration, statement, expression, etc., or more specific element of a statement, an expression, etc.
- A *Change operator* is one of the following in GumTree specifications: UPDATE, DELETE, INSERT, and MOVE.
- A *Repair action* represents a combination of a change operator and a code entity (e.g., `UPD stmt` or `DEL expr`).

[1] `http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/overview-summary.html`
[2] `http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Statement.html`
[3] `http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Expression.html`
[4] `https://github.com/GumTreeDiff/gumtree`
[5] `http://www.vogella.com/tutorials/EclipseJDT/article.html`

3.3 Research Questions

The objective of this study is to investigate the repair actions by closely looking at human-written patches, to build knowledge on which/how code entities are commonly involved/impacted by them. Our study examines the finer-grained AST diff representations of patches than the existing studies in the literature [157, 185, 254], to implement a closer look at code changes. In the study, we investigate the following research questions:

RQ1: Do patches impact some specific statement types? We revisit a common research question in the literature of patch mining studies: common statements changed by patches. In the majority of APR processes, the initial task is locating the buggy line or statement. Specifically, in generate-and-validate approaches, a spectrum fault localization technique (such as Tarantula [97], Ochiai [4], Op2 [177], Barinel [5] and DStar [232]) is used to identify suspicious lines or statements that are then mutated by APR tools [128, 224, 239, 240]. It is thus essential, based on real-world patches, to investigate which types of statements are recurrently involved in bug fix patches, and what kinds of repair actions are regularly applied to them by human developers.

RQ2. Are there code elements in statements that are prone to be faulty? A statement node in an AST representation can be decomposed into different children nodes whose types vary following the statement type. Consider the variable declaration statement "`private int id = 1;`". It can be decomposed into the modifier ("`private`"), the data type ("`int`"), the identifier ("`id`") and an initializer (the number literal "`1`"). Since common fault localization techniques can only point to suspicious lines, APR tools generally attempt to mutate the statements (often in a stochastic way) which can lead to nonsensical alien code [104]. With in-depth knowledge on fault-prone code elements of statements, APR tools can rapidly generate patch candidates that are more likely to be successful (with regards to the test cases), and which have more chances to be correct.

RQ3. Which expression types are most impacted by patches? Statements in Java programs are generally built based on expressions whose values eventually determine the execution behavior, and are often associated with bugs. In a preliminary study, we have found that in most patches, expressions were the buggy elements where a patch actually modifies a program. Our conjecture is that only a small number of expression types could be responsible for the majority of bugs at the expression level.

RQ4. Which parts of buggy expressions are prone to be buggy? Expressions in Java program can be composite entities: their AST nodes may have several children. For example, an `InfixExpression` consists of a left-hand expression, an infix operator, and a right-hand expression. We investigate the type of buggy elements within buggy expressions to further refine our understanding of recurring bug locations.

3.4 Study Design

We describe our dataset and the methodology for identifying bug fix patches and the code entities impacted by patches.

3.4.1 Dataset

For this study, we focus on Java open source projects commonly used by the research community [22, 157, 185, 254]. Table 3.1 enumerates the subject projects[6], of varying sizes, collected from the Apache software foundation[7]. These projects have been leveraged in previous studies on software patches

[6] `Lucene` and `Solr` share the same source code repository, so we put the results of the two projects in a single row.
[7] http://www.apache.org/index.html#projects-list

as they are reputed to provide commit messages that are clear and consistent with the associated code change diffs [254]. We further note that the Apache projects host an issue tracking system that is actively used, with a large number of commits keeping the link between reported bugs and the associated bug fix commits.

Table 3.1: Subjects used in our study. The number of bug fix commits actually used in the study is 16,450 as highlighted.

Projects	**LOC**	**# Commits**		
		All	Identified	**Selected**
commons-io	28,866	2,225	222	**191**
commons-lang	78,144	5,632	643	**522**
mahout	135,111	4,139	751	**717**
commons-math	178,84	7,228	1,021	**909**
derby	716,053	10,908	3,788	**3,356**
lucene-solr	943,117	51,927	11,408	**10,755**
Total	2,080,131	82,059	18,013	**16,450**

LOC: # lines of code. **All:** # of all available commits in a project. **Identified:** # of identified bug fix commits. **Selected:** # of selected bug fix commits actually used in this study.

3.4.2 Identification of Patches

We consider the following criteria of identifying bug fix commits in software repositories:

1. *Keyword matching*: we search commit messages for bug-related keywords (namely *bug*, *error*, *fault*, *fix*, *patch* or *repair*). This method was introduced by Mockus and Votta [171], and used in several studies [124, 185, 254].
2. *Bug linking*: we identify the reported and fixed bug IDs (e.g, **MATH-929**) in JIRA issue tracking system with two criteria: (1) *Issue Type* is '**bug**' and (2) *Resolution* is '**fixed**' [22]. This method entirely relies on the quality of links that developers maintain between their patches and the bug reports (*Type* is '**bug**' and *Resolution* is '**fixed**') in which they address the patch. We thus collect bug-fix commits by identifying such reported and fixed bug IDs in commit messages.

After applying the above criteria, we figure out that some selected commits are not actually bug fix commits; instead, they are commits regarding test cases, Javadoc and external documentation (e.g. xml files). These commits are out of scope and excluded from this study. Bug fix commits are collected by following the three criteria: (1) bug fix commits contains modified *.java* files, (2) these files do not have "*test*" in their names, and (3) these files can be parsed by GumTree to generate repair actions of buggy code fixing. Thus, 18,013 bug fix commits are collected from the seven projects. For example, in project `commons-io`, 403 and 185 commits are identified by *Keyword matching* and *Bug linking* respectively, but 222 and 138 of them are the commits that exactly contain changes in non-test Java files respectively.

To increase the confidence in our selected patches, we limit our study on patches with small-sized change hunks. The hunk size is defined as the number of lines of buggy code (respectively of fixed code) in a code change diff from a patch, where buggy hunk starts with '`-`' and fixed hunk starts with '+'. Figure 3.4 shows the distributions of sizes of buggy code hunks and fixed code hunks from all collected bug fix commits. In this study, the patches, whose buggy hunk size is up to 8 lines and fixed hunk size is up to 10, are selected as our dataset. These threshold values are set based on the Upper Whisker values from the hunk size distribution Tukey boxplots [63] in Figure 3.4[8]. To that end, 16,450 bug fix commits are selected as the data of this study.

[8]The upper whisker value are determined by 1.5 IQR (interquartile ranges) where IQR = 3rd Quartile − 1st Quartile, as defined in [63].

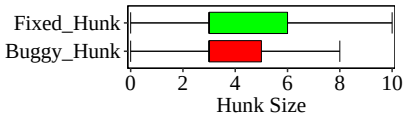


Figure 3.4: Distributions of buggy and fixed hunk sizes of collected patches. *Fixed/Buggy_Hunk* refers to fixed/buggy lines in a code change hunk of collected patches.

Previous studies have indeed shown that large code change hunks usually address feature addition, refactoring, etc. [85, 101], and do not often contain meaningful bug fix patterns [185]. Pan et al. [185] further reported that most bug fix hunks (91-96%) are small ones and ignoring large hunks has minimal impact on patch analysis.

3.4.3 Identification of Buggy Code Entities in ASTs

To identify the buggy code entities and their repair actions, all patches are parsed with GumTree [60]. The buggy code entities and their repair actions are identified by retrieving GumTree output in terms of its hierarchical construct. In this study, all elements of *deleted* and *moved* statements are treated as buggy code entities and all elements of *inserted* statements are treated as fixed code entities. For *updated* buggy statements, we further identify their exact buggy elements to find out the exact buggy code entities. For a buggy expression, if it is deleted, moved, or replaced by another expression, it is considered as a whole buggy expression. Otherwise, the buggy expression is further parsed to identify its buggy element(s).

3.5 Analysis Results

In this study, we investigate patches found in the seven projects listed in Table 3.1 to identify the distributions of buggy code entities and their corresponding repair actions. The results would answer the RQs described in Section 3.3. The distributions of the statistic data split by projects are similar to each other.

3.5.1 RQ1: Buggy Statements and Associated Repair Actions

Root AST node types in patches: Declaration entities in source code can also be buggy. Figure 3.5 provides a statistical overview of the root AST node types impacted by repair actions in patches. While statement entities occupy a large proportion in buggy code, it is noteworthy that buggy declarations, and associated repair actions are seldom mentioned in bug fix studies [157, 185, 254], and may thus be ignored by the APR community. Our study, however, finds that repair actions on declaration entities (i.e., *class* (`TypeDeclaration`), *enum*, *method* and *field* declarations) account for 26.7% of repair actions of patches, suggesting that the research community should put more efforts to investigate bugs related to declarations, as they may contribute to a significant portion of buggy code.

As shown in Figure 3.5, `TypeDeclaration` and `EnumDeclaration` only occupy 1.44% of repair actions of patches, that might be the reason why the state-of-the-art APR tools ignore the bugs relating these declaration entities and focus on fixing bugs at the statement level. However, buggy declaration entities indeed bother developers. For example, Figure 3.6 shows a patch of fixing bug MATH-927, a `TypeDeclaration`-related bug, which makes cloning broken and can cause `java.io.NotSerializableException`[9]. Thus it is fixed by adding the interface `Serializable` into

[9] `https://issues.apache.org/jira/browse/MATH-927`

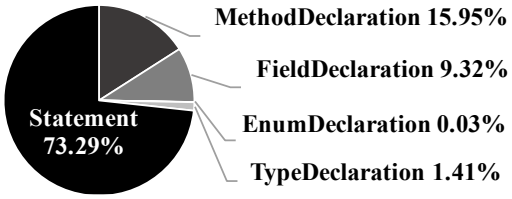


Figure 3.5: Distributions of root AST node types changed in patches.

```
1 Commit 185e3033ef43527a729c9dda5d57ed0537921a27
2 --- src/main/java/org/apache/commons/math3/random/BitsStreamGenerator.java
3 +++ src/main/java/org/apache/commons/math3/random/BitsStreamGenerator.java
4 @@ -29,2 +29,2 @@
5 public abstract class BitsStreamGenerator
6 -      implements RandomGenerator {
7 +      implements RandomGenerator, Serializable {
8
9 Repair actions parsed by GumTree:
10 UPD TypeDeclaration@@"BitsStreamGenerator"
11      INS Type@@"Serializable" to "TypeDeclaration"
```

Figure 3.6: Patch of fixing bug MATH-927, a `TypeDeclaration`-related bug, by adding the interface `Serializable`.

its `TypeDeclaration` node. This bug is also the bug `Math-12`[10] in benchmark Defects4J [98], however, it has not been fixed by any state-of-the-art APR tools yet[11], since those tools focus on the statement level to fix bugs.

Insight 1 ***Declaration entities in source code can also be buggy, and thus constitute a research opportunity for Automated Program Repair beyond statement level.*** *To fix bugs related to declaration entities, such as the bug in Figure 3.6, mutation-based tools (e.g., GenProg [224]) could generate mutations for the buggy* `TypeDeclaration` *by mutating common implementable interface types, pattern-based tools (e.g., PAR [104]) could summarize* `NotSerializableException` *fix pattern from this kind of patches, or search-based tools could specify constraints with fine-granularity information (e.g.,* `TypeDeclaration` *and* `NotSerializableException`*) to reduce search space and find fix ingredients from existing patches.*

Repair actions for statements: Statements (73.3% shown in Figure 3.5) are the main buggy code entities, which motivates researchers to fix bugs at the statement level. Therefore, to build the knowledge on repair actions at the statement level, we investigate the statement types impacted by patches as well as repair actions (categorized in *Update*, *Delete*, *Move* and *Insert*) that are applied to them. Figure 3.7 shows the distribution of statement types impacted by patches as well as the distributions of repair actions. The figure only lists up the top-5 statement types, the remaining are summed in an "Others" category.

1) Updating statements: As shown in Figure 3.7, a half of repair actions are statement updates, in which the entity types of buggy statements were not changed but their children entities were changed. This motivates researchers to fix bugs by mutating code at statement level (e.g., GenProg). However, coarse granularity is an important weak point for existing tools to fix bugs at the statement level.

Statements can be decomposed into several elements, which means that it would take a long time to generate patches by mutating each element even if it might succeed. For example, in Figure 3.1, the exact buggy code entity is the `InfixExpression` "`-dim / 2`", other code entities can interfere with

[10] `https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/12.src.patch`
[11] `http://program-repair.org/defects4j-dissection/#!/bug/Math/12`

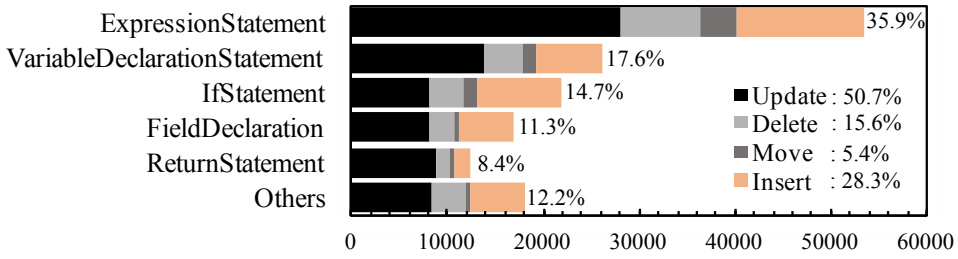


Figure 3.7: Distributions of statement-level repair actions of patches.

the mutating process of generating correct patches. Furthermore, the statement type can limit or noise the search space of finding fix ingredients. The buggy statement in Figure 3.1 is a `ReturnStatement`, which means this patch can only be a fix ingredient for buggy `ReturnStatement`s at the statement level. However, similar bugs can locate in other statement types (such as the bug in Figure 3.8).

```
1  double d = FastMath.pow(2 * FastMath.PI, -dim / 2) *
2             FastMath.pow(covarianceMatrixDeterminant, -0.5) *
3             getExponentTerm(vals);
4  return d;
```

Figure 3.8: A mutated bug of the bug in Figure 3.1.

Insight 2 ***The abundant real-world bugs fixed by updating buggy statements can support to tune the state-of-the-art APR tools by mining fine-grained characteristics with fine granularity (expression level) of patches.*** *For example, if APR tools could extract fine-grained context information of fixing the bug in Figure 3.1, such as the exact buggy* `InfixExpression` *"*`-dim / 2`*" and the detailed changes acted on the expression, to infer fix patterns (See Section 3.5.3) or constraint search space, they could fix more similar value-truncated bugs beyond the* `ReturnStatement` *code entity.*

2) Adoption of deletion and replacement: Simply deleting buggy statement(s) is an effective way of fixing bugs, which can also be combined with replacement. Recent experiments with APR techniques and program test suites have shown that some programs may pass tests when suspicious statements are simply deleted [224]. Thus, dropping buggy code statements could be attractive as the fast and effective way of fixing bugs. Our study shows that only 15.6% cases of repair actions consist of deleting buggy statements. Additionally, 45% of them, the code in the dropped statement(s) is inserted in another location to replace the dropped statement(s).

For example, Figure 3.9 shows that buggy code line is deleted but the buggy code expression is inserted in the new added `If` statement. Such patches are associated in the literature to the IF-related Addition of Post-condition Check (`IF-APTC`) fix pattern defined by Pan *et al.* [185], which is, however, in a high-level form and has not been used in APR tools.

Note that buggy code statement in Figure 3.9 is an `ExpressionStatement`. With further parsing of this bug, the exact buggy code entity is the `PrefixExpression` "`++objectiveEvaluation`". There are only 1,362 cases related buggy `PrefixExpression`s (See Table 3.3) that is a much smaller search space of fix ingredients for this bug compared against the statement level (more than 40,000 buggy `ExpressionStatement` cases, see the related value on the x-axis in Figure 3.7). If combining `ExpressionStatement` with `PrefixExpression`, we find that the search space can be further reduced to 28.

Insight 3 ***Expression-level granularity could improve the state-of-the-art APR tools by reducing search space, which could be further reduced if combining statement types with expression types.***

```
Commit e81ef196cd9dd3c7989b96f648f96ec138faa25b
--- src/java/org/apache/commons/math/optimization/general/AbstractLeastSquares
    Optimizer.java
+++ src/java/org/apache/commons/math/optimization/general/AbstractLeastSquares
    Optimizer.java
@@ -187, 1 +188, 4 @@
-   ++objectiveEvaluations;
+   if (++objectiveEvaluations > maxEvaluations) {
+     throw new FunctionEvaluationException(new
+     MaxEvaluationsExceededException(maxEvaluations), point);
+   }

Repair actions parsed by GumTree:
DEL ExpressionStatement@@"++objectiveEvaluations;"
INS IfStatement@@"if"
   INS InfixExpression@@"code" to IfStatement
     MOV PrefixExpression@@"++objectiveEvaluations"
         to InfixExpression
   INS ......
```

Figure 3.9: An infinite-loop bug taken from project `commons-math` is fixed by replacing buggy statement type.

3) Moving statements: Moving a buggy statement(s) to correct its position is another effective way of fixing bugs. We observe that 5.4% of repair actions involve moving statements across the program code. Figure 3.10 shows an example of fixing a bug by moving the buggy statement to the correct position. It is difficult to obtain valuable information from its simple repair action, but its context information, such as its parent statement (i.e. `WhileStatement`) and the dependency of three variables (i.e., `start`, `end`, and `ranges`), could be used to tune APR tools.

```
Commit 9c5be23a3d00b4238ddb3794a1ffec463f2ceac9
--- solr/core/src/java/org/apache/solr/cloud/HashPartitioner.java
+++ solr/core/src/java/org/apache/solr/cloud/HashPartitioner.java
@@ -46, 5 +55, 5 @@
   while (end < Integer.MAX_VALUE) {
      end = start + srange;
-     start = end + 1L;
      ranges.add(new Range(start, end));
+     start = end + 1L;
   }

Repair actions parsed by GumTree:
MOV ExpressionStatement@@"start = end + 1L;"
    from 1 to 2 in WhileStatement_BodyBlock
```

Figure 3.10: A bug taken from project `solr` is fixed by moving the buggy statement.

Insight 4 ***To generate fix patterns or create search space with patches involving "move" actions, more context information should be considered.***

4) Recurrently impacted statements: A few statement types are recurrently impacted by patches. From the distribution of statement types in Figure 3.7, we note that 5 (out of 22) statement types (namely `ExpressionStatement`, `VariableDeclarationStatement`, `IfStatement`, `ReturnStatement` and `FieldDeclaration`) represent ∼88% of statements impacted by patches. These statistics support the motivation of many researchers to focus on repairing a specific type of statements: ACS [239] is such an example APR technique targeting `IfStatement`-related bugs. Our study highlights other statement types which can benefit from targeted approaches. In particular, `ExpressionStatement` is

impacted by a third (∼36%) of repair actions, suggesting that statements of this type are more likely to contain bugs than other types of statements.

3.5.2 RQ2: Fault-prone Parts in Statements

As discussed in Section 3.5.1, if fine-granularity information can be extracted from existing patches, it could improve APR tools. Fortunately, statements can be decomposed into different sub elements, which supports us to further investigate exact buggy elements of statements. To the best of our knowledge, we are the first to take a close look at real-world patches in finer granularity than statement level in the literature. Our study may yield further insights into which code entities, beyond whole statements, are recurrently buggy and should thus be the focus of APR techniques.

A statement node in an AST representation can be decomposed into several children elements. In this study, all elements of statements are classified into four categories: *Modifier*, *Type*, *Identifier*, and *Expression* where *Modifier* denotes the modifiers of source code in its AST. *Type* refers to any type nodes, *Identifier* can be a name of *class*, *method*, or *variable*, and *Expression* includes the 35 kinds of expressions defined in the Eclipse JDT APIs. The statistic distributions of these elements impacted by patches are provided in Figure 3.11.

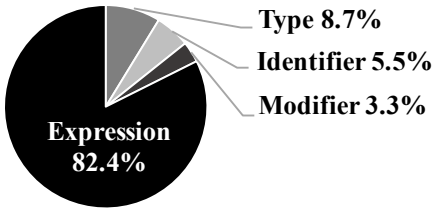


Figure 3.11: Distributions of inner-statement elements impacted by patches.

Patches for "modifier" bugs: *Modifier* elements in statements can be buggy, and repair actions associated with them have simple instructions. Figure 3.11 presents that 3.3% cases of repair actions fixing bugs involve *Modifier* elements (i.e., qualifiers such as `public`, `final`, `static`). The bugs in such cases often are caused by missing a necessary modifier or assigning an inappropriate one. At best, these bugs can create style mismatch in the code, and at worst, can present semantic implications for program behavior. We can enumerate three ways repair actions that are applied:

1. *Add a missing modifier*, as in patch *A* of Figure 3.12, where the missed modifier "`volatile`" is inserted to make the variable "`defaultStyle`" thread-safe.
2. *Delete a redundant modifier*, as in patch *B* of Figure 3.12, where redundant modifier "`final`" is dropped to avoid exposing a mutating map.
3. *Replace an inappropriate modifier*, as in patch *C* of Figure 3.12, where modifier "`protected`" is changed into "`private`" to prevent potential vulnerability.

The Java language supports 12 *Modifier* types[12] whose inappropriate usage could lead to bugs and even vulnerabilities. The FindBugs [90] static analyzer even enumerates 17 bug types related to modifiers. Actually, four projects in our study integrate Findbugs in their development chain[13,14,15,16]. However, fixing those modifier-related bugs in these projects are still addressed manually. Additionally, all modifier-related bugs in benchmark Defects4J have not been fixed by any state-of-the-art APR tools[17] The reason might be that APR tools cannot fix modifier-related bugs because of coarse granularity. So that it is necessary to tune APR tools to fix modifier related bugs with finer granularity.

[12] `https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Modifier.html`
[13] `https://github.com/apache/commons-io/blob/master/pom.xml#L373`
[14] `https://github.com/apache/commons-lang/blob/master/pom.xml#L685`
[15] `https://github.com/apache/commons-math/blob/master/pom.xml#L717`
[16] `https://github.com/apache/mahout/blob/master/pom.xml#L771`
[17] `http://program-repair.org/defects4j-dissection/#!/`

```
A: an example of adding a missed modifier:
Commit d55299a0554375f3f935a0ff85bd2002faa2ec55 (LANG-487)
--- src/java/org/apache/commons/lang/builder/ToStringBuilder.java
+++ src/java/org/apache/commons/lang/builder/ToStringBuilder.java
@@ -97, 1 +97, 1 @@
-   private static ToStringStyle defaultStyle =
+   private static volatile ToStringStyle defaultStyle =
                                       ToStringStyle.DEFAULT_STYLE;

Repair actions parsed by GumTree:
UPD FieldDeclaration
    INS Modifier@@"volatile" to FieldDeclaration

B: an example of deleting a redundant modifier:
Commit 60fe05eb7a32a4a178f4212da6a812c80cedce82 (LANG-334)
--- src/java/org/apache/commons/lang/enums/Enum.java
+++ src/java/org/apache/commons/lang/enums/Enum.java
@@ -305, 1 +305, 1 @@
-   private static final Map cEnumClasses = new WeakHashMap();
+   private static Map cEnumClasses = new WeakHashMap();

Repair actions parsed by GumTree:
UPD FieldDeclaration
    DEL Modifier@@"final" from FieldDeclaration

C: an example of replacing the inappropriate modifier:
Commit 0e07b8e599e0d59c259dcc8501167a80d030179b
--- src/java/org/apache/commons/lang/builder/EqualsBuilder.java
+++ src/java/org/apache/commons/lang/builder/EqualsBuilder.java
@@ -111, 1 +111, 1 @@
-   protected boolean isEquals;
+   private boolean isEquals;

Repair actions parsed by GumTree:
UPD FieldDeclaration
    UPD Modifier@@"protected" to "private"
```

Figure 3.12: Three bugs in project `commons-lang` are fixed by changing their modifiers.

Insight 5 ***The small number of modifier types allows researchers to enumerate all possible mutations or change patterns for buggy modifier(s) at modifier level, and to reduce search space of fix ingredients for modifier-related bugs.*** *Additionally, existing patches of fixing modifier-related bugs and the static analysis tools (e.g., FindBugs provides detailed definitions for specific modifier-related bugs) can help researchers tune APR tools to fix specific modifier-related bugs automatically, like the thread-safe bug of patch* A *in Figure 3.12.*

Fixing modifier-related bugs can, however, have unsuspected impacts beyond the code base, and thus may constitute a new height that APR techniques can try to reach through patch prioritization. For example, a fix may break the backward compatibility of client applications. Figure 3.13 illustrates an example of a modifier-related patch in the project `Apache ant`[18], but which leads to that the value of `systemClasspath` cannot be re-assigned in Eclipse, so that it breaks the Eclipse integration[19]. Therefore, *fixing modifier-related bugs should consider the backward compatibility of software.*

Patches for "Type" nodes: *Type* code entities can also be buggy, and their fix ingredients may be specific. In this study, *Type* refers to the data type in the code, such as "`int`" in Figure 3.14. Changes applied to *Type* nodes account for 8.7% cases of repair actions. The bug in Figure 3.14 is

[18] `http://ant.apache.org/`
[19] `https://bz.apache.org/bugzilla/show_bug.cgi?id=60582`

```
Commit 984a03d1ceb6e4b5d194e4d639d0b0fca46d92be
--- src/main/org/apache/tools/ant/types/Path.java
+++ src/main/org/apache/tools/ant/types/Path.java
@@ -70, 2 +70, 2 @@
-   public static Path systemClasspath =
+   public static final Path systemClasspath =
            new Path(null,System.getProperty("java.class.path"));

Code @Line 1484 in InternalAntRunner.java in Eclipse project:
org.apache.tools.ant.types.Path.systemClasspath = systemClasspath;
```

Figure 3.13: A modifier-related patch breaks the backward compatibility of project `Apache ant`.

```
Commit b032fb49f09c020174da1d5d865d878b8351d89d
--- lucene/codecs/src/java/org/apache/lucene/codecs/compressing/CompressingSto
    redFieldsIndex.java
+++ lucene/codecs/src/java/org/apache/lucene/codecs/compressing/CompressingSto
    redFieldsIndex.java
@@ -366,1 +366,1 @@
-      int startPointer = 0;
+      long startPointer = 0;

Repair actions parsed by GumTree:
UPD VariableDeclarationStatement
    UPD Type@@"int" to "long"
```

Figure 3.14: An integer-overflow bug taken from project `lucene` is fixed by modifying the variable data type.

an integer-overflow bug taken from project `lucene` and fixed by replacing the data type "`int`" with "`long`".

Insight 6 ***Theoretically, repair of such traditional programming bugs can be performed readily by APR tools when key context information is available.*** *For example, if the nature of the bug (e.g., "*integer-overflow*") is known, the APR tool could attempt, as one of its fix rules/templates, to replace the type "`int`" with "`long`" that has a bigger memory size. If GenProg or PAR could learn repair actions from this kind of patches to mutate the type nodes of bugs but not the whole statements, which could fix similar bugs like Math_30 and Math_57 in Defects4J that have not been fixed by these tools.* Nevertheless, the challenge arises for APR tools when the buggy type is a specific data type, which requires more precise context information.

Patches for "Identifier" nodes: Identifiers are also impacted by patches, and naming an appropriate identifier is not an easy task. Similarly, changes applied to *identifiers* involve in 5.5% cases of repair actions. Changes on identifiers are generally about assigning appropriate names to identifiers to avoid confusion or inadequate usages which often complicate maintenance tasks or even lead to bugs [1, 36]. Thus, we can enumerate two ways in which repair actions are applied:

1. *Modifying identifiers to satisfy naming convention*, as in patch A of Figure 3.15, where the old identifier of field '`random`' is changed to 'RANDOM' by re-writing it with upper-case letters since constant names should be in upper-case letters recommended by Java naming conventions[20].
2. *Modifying inconsistent identifiers*, as in patch B of Figure 3.15, where the old variable name "`result`" is replaced with a new name "`recommendations`" which seems to be more consistent and easier to track during maintenance.

This kind of changes may be questioned as bug fixes, but we find some of them are linked to bug reports (e.g., `MAHOUT-1151` in Figure 3.15). Naming things is the hardest task that programmers have

[20] `http://www.oracle.com/technetwork/java/codeconventions-135099.html`

```
A: Example of incorrect naming convention identifier changes:
Commit ad2817beb235f8f24b7e73feac2ad717346bcd6f
--- core/src/main/java/org/apache/mahout/clustering/dirichlet/UncommonDistributions.java
+++ core/src/main/java/org/apache/mahout/clustering/dirichlet/UncommonDistributions.java
@@ -31, 1 +31, 1 @@
-    private static final Random random = RandomUtils.getRandom();
+    private static final Random RANDOM = RandomUtils.getRandom();

Repair actions parsed by GumTree:
UPD FieldDeclaration
    UPD SimpleName@@"random" to "RANDOM"

B: Example of inconsistent identifier changes:
Commit c3154b86dce55f8ca318c35f97751d3aeee415aa (MAHOUT-1151)
--- core/src/main/java/org/apache/mahout/cf/taste/hadoop/als/RecommenderJob.java
+++ core/src/main/java/org/apache/mahout/cf/taste/hadoop/als/RecommenderJob.java
@@ -121, 1 +124,1 @@
-    private RecommendedItemsWritable result =
+    private RecommendedItemsWritable recommendations =
                                    new RecommendedItemsWritable();

Repair actions parsed by GumTree:
UPD FieldDeclaration
    UPD SimpleName@@"result" to "recommendations"
```

Figure 3.15: Two identifier changes taken from project `mahout`.

to do [96], thus it is inevitable to generated bugs because of inconsistent identifiers [1, 36]. FindBugs also enumerates 10 bug types related to identifiers. So far, a number of research directions on related to identifiers in code have been explored in the literature: Høst and Østvold [89] used name-specific implementation rules and certain semantic profiles of method implementations to find and fix method naming bugs, but limited to method names starting with "`contain`" or "`find`". Kim et al. [108] relied on a custom code dictionary to detect inconsistent identifiers. Allamanis et al. [9, 10, 12] leveraged deep learning techniques to suggest identifiers for variables, methods, and classes with sub-tokens extracted from code.

Although, current research contributions have shown promising results about identifier-related studies, identifying and fixing inconsistent identifiers remains an open challenge because of their shorting comings, such as inadequate context information.

Insight 7 ***Identifiers are the basic knowledge of code understanding, thus, more context information (e.g., method implementation should be considered to name method identifiers) should be considered to address fixing inconsistent identifiers.*** *Changing identifiers, however, is not a trivial endeavor: it may break the backward compatibility of applications, and developers' understanding of code might be impacted by identifier changes. This challenge may thus be a relevant and worthy target for APR research.*

Patches for "Expression" nodes: Expression is the main fault-prone element of statements. We observe that *Expressions* are concerned by 82% cases of repair actions. Statements in Java program are generally built based on various expressions whose values eventually determine the execution behavior. It is thus reasonable that most bugs are associated with expressions. Therefore, it does not come as a surprise that the majority of repair actions in patches are performed to mutate expressions. As 35 different expression types are defined in Eclipse JDT APIs, and many of them can

be decomposed in several elements, we will take a close look at their repair actions in more details in following sections.

3.5.3 RQ3: Buggy Expressions and Associated Repair Actions

We further investigate which kinds of expressions are recurrently impacted by repair actions on code statements by retrieving the sub-trees of buggy statements to find the exact buggy expressions. For example, in the AST sub-tree (illustrated in Figure 3.3) of the buggy statement in Figure 3.1, the `InfixExpression` "`FastMath.pow(2 * FastMath.PI, -dim / 2) * FastMath.pow(covarianceMatrixDeterminant, -0.5) * getExponentTerm(vals)`" is impacted by this patch. With further parsing, the `MethodInvocation` "`FastMath.pow(2 * FastMath.PI, -dim / 2)`" is the more exact expression impacted by this patch than its parent infix-expression. Finally, we can find that the exact buggy expression is the `InfixExpression` "`-dim / 2`". All hierarchical expressions (i.e., `InfixExpression` → `MethodInvocation` → `InfixExpression`), eventually lead to the exact buggy code "`-dim / 2`", are obtained by looking closely into the AST sub-tree of the buggy statement.

The distributions of expression types impacted by patches are presented in Figure 3.16. Figure 3.16 only lists up top-5 expression types. The remaining are summed in an "Others" category.

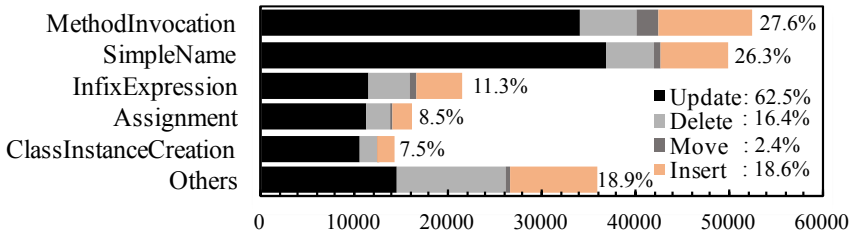


Figure 3.16: Distributions of repair actions at the expression level.

Repair actions of recurrently impacted expressions: A small number of expression types are recurrently impacted by patches. It is noteworthy that the 5 out of 35 expression types (namely `MethodInvocation`, `SimlpeName`, `InfixExpression`, `Assignment`, and `ClassInstanceCreation`) account for ∼81% cases of repair actions at the expression level. In particular, repair actions on `MethodInvocation` and `SimpleName` account for more than half of repair action cases. In this study, `MethodInvocation` expressions are method references, and `SimpleName` expressions denote variable names and method names. Their presence indicates that incorrect references to methods and variables are the main cause of many bugs.

Insight 8 ***A small number of expression types are recurrently impacted by real-world patches, which can support the motivation that mutation-based APR tools (e.g., GenProg) generate mutations and pattern-based APR tools (e.g., PAR) mine fix patterns for a specific type of expressions.*** *For example, in Figure 3.1, the exact buggy expression is an* `InfixExpression`*: "*`-dim / 2`*", and is fixed by replacing it with another* `InfixExpression`*: "*`-0.5 * dim`*". It is known that "*`1.0 / 2 = 0.5`*" can represent the relationship between the deleted* `NumbeLiteral "2"` *and the inserted* `NumberLiteral "0.5"`*, further inferred that "*`-1.0 / 2 * dim`*" is a function-identical mutation of the patch code "*`-0.5 * dim`*". With the following inferring process, it is easy to extract a fix pattern for value-truncated bugs at the expression level beyond the limitation of statement types.*

$$\begin{aligned} &dim/2 \;\rightarrow\; 0.5 * dim \;\rightarrow\; 1.0/2 * dim \\ &\Rightarrow Pattern : a/b \rightarrow 1.0/b * a, (a : dividend, b : divisor) \end{aligned} \tag{3.1}$$

```
Commit 37ecb1c20f0ed36e7c438d265b0c30a282e4fff5
--- lucene/core/src/java/org/apache/lucene/store/Directory.java
+++ lucene/core/src/java/org/apache/lucene/store/Directory.java
@@ -200, 1 +200, 1 @@
-   is.copyBytes(os, is.length());
+   os.copyBytes(is, is.length());
Repair actions parsed by GumTree:
UPD ExpressionStatement@@"is.copyBytes(os,is.length());"
    UPD MethodInvocation@@"is.copyBytes()"
        UPD SimpleName@@"is" to "os"
            UPD SimpleName@@"os" to "is"
```

Figure 3.17: Bug LUCENE-4377 is fixed by modifying the wrong `SimpleName` expressions "`is`" and "`os`".

However, it is difficult to mine fix patterns only with the buggy `SimpleName` expressions since they capture less useful characteristics. For example, Figure 3.17 shows a bug is fixed by modifying the buggy `SimpleName is` and `SimpleName os` that are meaningless or could be any identifiers, so that it is difficult to extract distinguishing characteristics from them. Therefore, *if mining fix patterns from patches involving `SimpleName` expression changes, more context information (such as its method reference "`copyBytes`") should be considered.*

Rarely impacted expressions: There are expression entities rarely changed by patches. It is also noteworthy that there are very few cases (less than 0.05%, 100 cases) of repair actions involving `LambdaExpression`, `CharacterLiteral`, `TypeLiteral`, `Annotation` and `SuperFieldAccess` expressions. Our data also includes no repair action case impacting `ExpressionMethodReference`, `MethodReference`, `SuperMethodReference`, and `TypeMethodReference`. In the case of `LambdaExpression`, `CreationReference`, `ExpressionMethodReference`, `MethodReference`, `SuperMethodReference`, and `TypeMethodReference`, we understand that they have been introduced in Java 8[21], and are thus not yet involved in bugs from our dataset. *It implies that APR tools could ignore such expressions when fixing bugs.*

Table 3.2: Distributions of repair actions on buggy literal expressions.

Expressions	Updated	Deleted or replaced by other expressions
BooleanLiteral	12%	88%
CharacterLiteral	46%	54%
NumberLiteral	65%	35%
StringLiteral	62%	38%

Repair actions of literal expressions: Literal expressions can also lead to bugs, and their repair actions could be specific. Table 3.2 presents the distributions of repair actions on buggy literal expressions. Recurrent repair actions on `BooleanLiteral` expressions are mostly deleting them or replacing them with other types of expressions. We note that in only a few cases, the repair action switches "`true`" and "`false`" booleans. In the case of buggy `CharacterLiteral` expressions, the related repair actions are balanced on updating the literal values and deleting them or replacing them with other types of expressions. Both buggy `NumberLiteral` and `StringLiteral` expressions have similar distributions of repair actions. Ratios of updating the buggy values are slightly higher than other repair actions. `StringLiteral` related bugs can be very specific, such as the bug in Figure 3.18, thus, *more specific context information or fix ingredients are needed to fix literal expression related bugs, which arises a new height challenge for APR tools.*

[21] `http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`

```
Commit ae4734f4cfc17453f8d5889a08ae90bb6d3601b7
--- solr/core/src/java/org/apache/solr/util/SimplePostTool.java
+++ solr/core/src/java/org/apache/solr/util/SimplePostTool.java
@@ -778, 1 +778, 1 @@
-    if (type.equals("text/xml")
+    if (type.equals("application/xml")
        || type.equals("text/csv") || type.equals("application/json")) {
Repair actions parsed by GumTree:
UPD IfStatement@@"If"
    UPD InfixExpression@@"InfixExp_code"
        UPD MethodInvocation@@"type.equals()"
            UPD StringLiteral@@"text/xml" to "application/xml"
```

Figure 3.18: Patch of fixing bug SOLR-6959 (`StringLiteral`-related bug).

3.5.4 RQ4: Fault-prone Parts in Expressions

In this section, we further investigate the distributions of buggy sub-elements of expressions. Our investigation results are provided in Table 3.3. The first column of Table 3.3 enumerates different expressions types. The second column represents the percentage in which each expression is replaced or deleted as a whole. An expression can be further decomposed into several sub-elements. For example, an `InfixExpression` consists of a left-hand expression, an infix operator, and a right-hand expression. The third column shows the percentage in which each sub-element is changed.

Table 3.3: Distribution of *whole* vs. *sub-element* changes in buggy expressions.

Expression	Quantity	% whole expression	% each sub-element		
ArrayAccess	1,127	47.7%	ArrayExp(35.4%)	ArrayIndex(20.6%)	
ArrayCreation	740	27.3%	ArrayType(14.2%)	Initializer(60.9%)	
Assignment	13,762	18.1%	Left_Hand_Expression(13.3%)	Operator(0.8%)	Right_Hand_Expression(73.5%)
CastExpression	2,192	45.8%	Type(11.9%)	Expression(42.9%)	
ClassInstanceCreation	12,385	15.5%	Expression(10.2%)	ClassType(19.7%)	Arguments(63.0%)
ConditionalExpression	882	22.9%	Condition_Expression(24.1%)	Then_Expression(33.0%)	Else_Expression(49.5%)
FieldAccess	568	57.2%	Expression(9.2%)	Field(35.9%)	
InfixExpression	15,896	27.3%	Left_Hand_Expression (35.0%)	Operator(5.6%)	Right_Hand_Expression(68.7%)
InstanceofExpression	371	55.5%	Expression(16.7%)	Type(30.5%)	
MethodInvocation	40,054	14.7%	MethodName(22.1%)	Arguments(79.8%)	
PostfixExpression	512	85.2%	Expression (14.6%)	Operator(0.8%)	
PrefixExpression	1,362	50.0%	Operator (0.1%)	Expression (49.9%)	
QualifiedName	4,567	48.7%	QualifiedName (10.0%)	Identifier(48.9%)	
VariableDeclarationExpression	676	67.3%	Modifier (32.7%)		

† "% whole expression" indicates the percentage in which the whole buggy expression is replaced by another expression or removed directly. "% sub-elements" represents the percentage in which one or more sub-elements of an expression are changed instead of the whole expression. For each expression type, the sum of percentages may not be 100% since sub-expressions in the third column can be overlapped among each other. For example, for the *ArrayAccess* expression, the sum percentage of *ArrayExp* and *ArrayIndex* is 81.8% in linked patches that is over 74.7% (100%−35.6%), which indicates that both *ArrayExp* and *ArrayIndex* of some buggy *ArrayAccess* expressions are changed simultaneously in same bug fixes. The same as other expressions.

Faulty parts of expressions: Not all parts of the expressions are completely faulty, but some specific sub-elements are the exact buggy parts. As shown in Table 3.3, there are different percentages of fault-prone parts for each expression type, which provides an abundant resource of learning fix behavior for various specific bugs. For example, the whole buggy expressions could improve APR tools by reducing search space to find fix ingredients by combining their parent statement types, such as the bug fix shown in Figure 3.9 and Insight 3.

Insight 9 ***The statistics can support to categorize bug fixes, mine fix patterns mining, or reduce search space at expression level with common distinguishing characteristics (such as non-faulty parts of expressions) to tune APR tools.*** *For example, the exact buggy entity in Figure 3.19 is the* `MethodInvocation` *"*`value.toUpperCase()`*", which can cause i18n issues*[22]*. because of the missing parameter "*`Locale.ROOT`*". Where "*`value`*", a* `String`*-type valuable, and "*`toUpperCase`*", a* `String` *related API, are the non-faulty elements and specific characteristic of this bug fix, which can be used to classify the bug as a string letter-case transforming bug "str.toUpperCase()".*

[22] `https://garygregory.wordpress.com/2015/11/03/java-lowercase-conversion-turkey/`

With the corresponding repair actions, an executable fix pattern (as below) can be extracted. Method name "`toUpperCase`" can also be the specific constraints to search fix ingredients for the i18n issues.

$$str.toUpperCase() \rightarrow str.toUpperCase(Locale.ROOT) \tag{3.2}$$

```
Commit 44854912194177d67cdfa1dc765ba684eb013a4c
--- src/main/java/org/apache/commons/lang3/time/FastDateParser.java
+++ src/main/java/org/apache/commons/lang3/time/FastDateParser.java
@@ -895, 1 +895, 1 @@
-  final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase());
+  final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase(Locale.ROOT));

Repair actions parsed by GumTree:
UPD VariableDeclarationStatement@@"code"
   UPD VariableDeclarationFragment@@"tz"
      UPD MethodInvocation@@"TimeZone.getTimeZone()"
         UPD MethodInvocation@@"value.toUpperCase()"
            INS QualifiedName@@"Locale.ROOT" to MethodInvocation
```

Figure 3.19: Patch of bug LANG-1357 fixed by adding the parameter "`Locale.ROOT`" into the `MethodInvocation`: "`toUpperCase()`".

Non-recurrent faulty operators: Faulty operators are not recurrent in real-world bugs. It is noteworthy that repair actions on operators only account for 0.7% of all repair actions for expressions. Specifically, fixing operators only account for 0.8% cases in buggy `Assignment` expressions, 5.6% cases in `InfixExpressions`, 0.8% cases in `PostfixExpressions`, and 0.1% cases in `PrefixExpressions`. *It implies that when APR tools generate mutations to fix bugs, they should focus on non-operator code entities of potential buggy code.*

3.6 Threats to Validity

A threat to validity is the complexity of patches. Patches could involve updating `MethodDeclarations`, and most repair actions on `MethodDeclarations` (except for repair actions on its *Modifier* and *Identifier*) lead to the changes of method bodies, which further complicates accurate modeling or learning of the repair actions. Patches about adding new methods or code files, patches with multi-hunk changes or several code files challenge fix behavior learning and pattern mining. To reduce this threat, we select patches with small size hunks. Threats to validity include the limitations of the underlying tool (GumTree) used in this study. GumTree may produce unfeasible edit scripts. We add new labels into GumTree to reduce this threat, which could be further reduced by developing more advanced tools. Threats to validity also include the limitation of identifying bug fix commits. To reduce this threat, in this study, bug fix commits are collected in two different ways.

3.7 Related Work

Bug fix commits study: Various studies have mined software repositories to analyze commits associating patches. Purushothaman and Perry [192] studied patch-related commits in terms of sizes of bug fix hunks and repair action types (i.e., update, insert and delete) to investigate the impact of small source code changes. German [66] analyzed the characteristics of *modification records* (i.e., source code changes in the version control system of software) from three aspects: authorship, the number of files, and modification coupling of files. Alali et al. [8] analyzed the relationships among three size metrics (# of files, # of lines, and # of hunks) for commits to infer the characteristics of

commits from years of historical information. Yin et al. [247] presented a comprehensive characteristic study on incorrect bug-fixes which are figured out by tracking the revision history of each patch, and showed that bug fixes could further cause new bugs. Thung et al. [218] performed a study on real faults to investigate whether bugs are localizable by extracting faults from code changes manually. Their results showed that most faults are not within small code hunks. Nguyen et al. [178] studied the recurrent code changes and found that repetitiveness is common in bug fix hunks with small size. Eyolfson et al. [59] investigated the relationship between time-based characteristics of commits and their bugginess, of which results showed that the bugginess of a commit is correlated with the commit time. However, these studies did not investigate the links between the nature of bug fixes and automatic program repair, which is analyzed in this study.

Patches study: Pan et al. [185] manually explored 27 common bug fix patterns in Java programs to understand how developers change code to fix bugs. Martinez et al. [157] and Zhong et al. [254] analyzed the repair actions of patches at the statement level to understand the nature of bugs and patches. Although these studies provide interesting insights into program repair, they could be misleading for implementing automated repair actions because of the coarse-grained level of statements. As listed in Table 3.4, the three studies focus on statement level to investigate patches. Indeed, as investigated in this study, buggy parts can be localized in a more fine-grained way, which could lead to more accurate repair actions. Last but not least, moving buggy statement is also an effective way of fixing bugs, which is, however, ignored by them.

Table 3.4: Comparison of our work with other previous real-world patch studies.

Patch study	Granularity of code entities	Granularity of change operators
Pan et al. [185]	Statement level.	Abstract patterns.
Martinez et al. [157]	Statement level and method invocations.	Update, delete, and insert.
Zhong et al. [254]	Statement level.	Modify, add, and delete.
Our work	All AST node code entities impacted by patches.	Update, delete, move, and insert.

Program repair with real-world patches: Kim et al. [85] proposed PAR which utilizes common fix patterns to automatically fix bugs. Le et al. [124] extended PAR by automatically mining bug fixes across projects in their commit history to guide and drive a program repair. Bissyande [29] considered also investigating fix hints for reported bugs. Tan et al. [213] analyzed anti-patterns that may interfere with the process of automated program repair. Long et al. [147] proposed a new system, Genesis, that processes patches to automatically infer code transforms for automated patch generation. These studies obtained promising results, but they have a common limitation that focuses on statement level but not as the finer granularity at expression level investigated in this study.

3.8 Summary

Real-world patches can provide useful information (e.g., on repair actions) for learning-based and template-driven automated program repair techniques, allowing for fast generation of correct patches. In general even, we argue that towards boosting the performance of automated program repair techniques, the community needs to deepen its knowledge on bug fix code transformations from real-world (i.e., human-written) patches. In this study, we engaged in this endeavor through a systematic and fine-grained investigation of 16,450 bug fix-related commits collected from seven open source Java projects. We find that there are opportunities for APR techniques to be targeted at code elements that have not yet been investigated. We also find that a small number of statement and expression types are recurrently impacted by real-world patches, and expression-level granularity could reduce search space of finding fix ingredients for similar bugs. We further discuss nine insights into tuning APR tools, challenges and possible resolves through investigating research questions around the actual locations of buggy code and repair actions at the AST level.

4 An Investigation of Fault Localization Bias in Benchmarking APR Systems

In this work, we identify and investigate a practical bias caused by the fault localization (FL) step in a repair pipeline. We propose to highlight the different fault localization configurations used in the literature, and their impact on APR systems when applied to the Defects4J benchmark. Then, we explore the performance variations that can be achieved by "tweaking" the FL step. Eventually, we expect to create a new momentum for (1) full disclosure of APR experimental procedures with respect to FL, (2) realistic expectations of repairing bugs in Defects4J, as well as (3) reliable performance comparison among the state-of-the-art APR systems, and against the baseline performance results of our thoroughly assessed kPAR repair tool. Our main findings include: (a) only a subset of Defects4J bugs can be currently localized by commonly-used FL techniques; (b) current practice of comparing state-of-the-art APR systems (i.e., counting the number of fixed bugs) is potentially misleading due to the bias of FL configurations; and (c) APR authors do not properly qualify their performance achievement with respect to the different tuning parameters implemented in APR systems.

This chapter is based on the work published in the following research paper:

- Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, pages 102–113. IEEE, 2019

Contents

4.1 Overview

Automated program repair (APR) holds the promise of reducing manual debugging effort by automatically generating patches for defects identified in a program. In production, APR will drastically reduce time-to-fix delays and limit downtime. In a development cycle, APR can help suggest changes to accelerate debugging. In the literature, there are two distinct repair scenarios: (1) fixing *syntactic errors*, i.e., cases where code violates some programming language specifications [27, 75] and (2) fixing *semantic bugs*, i.e., cases where implementation of program behaviour deviates from developer's intention [163, 179]. The latter requires Fault Localization (FL) through execution of test cases. It is the scope of this paper.

Once a fault is arosen, most recent APR systems follow the same basic pipeline as shown in Figure 4.1: (1) fault localization (FL), (2) patch candidate generation, and (3) patch validation. The FL step identifies an entity in a program as the potential fault location. In patch generation, given a fault location, the APR system modifies the program, i.e., creates a patch. The last step assesses whether the patch actually fixes the defect. If the patch is not regarded as a valid patch, the second and last steps are repeated until a valid patch is generated or the termination condition is satisfied. To increase the chances of finding a valid patch, the process is iterated over all suspicious code locations ranked by FL tools.

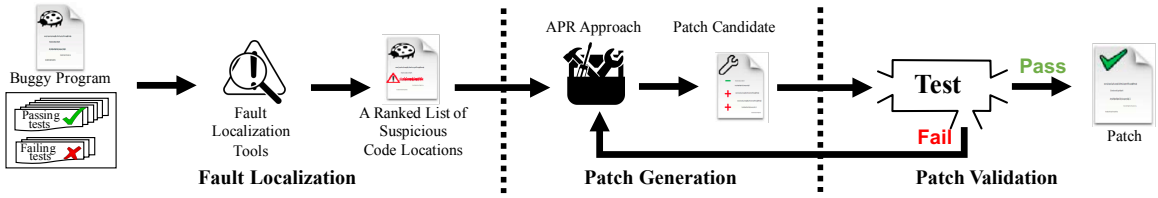


Figure 4.1: Standard steps in a pipeline of Automated Program Repair.

In the repair pipeline, APR systems generally focus on the patch generation step, but tend to use similar strategies for fault localization and patch validation. To the best of our knowledge, most of the current state-of-the-art APR approaches [43, 47, 102, 104, 122–124, 128, 141, 147, 148, 150, 164, 179, 224, 239, 240] leverage test suites to perform fault localization and patch validation. For fault localization, the systems rely on a testing framework such as GZoltar [40], and a spectrum-based fault localization formula [234, 241, 252], such as Ochiai [4]. Eventually, bug fixing performance is measured by counting the number of bugs for which the system can generate a patch that passes all test cases. Such patches are claimed to be valid.

Nevertheless, given the growing interest in APR among software engineers, it is important to ensure that the research outputs are relevant and well assessed in terms of reliable performance for practitioners. In this respect, the APR research community has already started to reflect on the *acceptability* [104, 173] and *correctness* [202, 238] of the patches generated by APR tools. Researchers [30, 125, 194, 202, 245] raised the concern of overfitting patches: those are generated patches that can pass the validating test cases, but may actually not be the semantically-correct patches for repairing the defect.

Since then, assessment of APR approaches in the literature attempts to provide information on the number of generated patches that are *plausible* (i.e., they make the programs pass all the test cases) and the number of patches that are *correct* (i.e., they are equivalent to the patches that were actually submitted by the program developers). Table 4.1 provides an example of assessment results excerpted from the paper describing SimFix [93], one of the most recent state-of-the-art works on APR that was tested on the Defects4J [98] programs. Based on data reported in this table, researchers explicitly rank the APR systems, and use this ranking as a validation of new achievements in program repair.

Table 4.1: Table excerpted from [93] with the caption "*Correct patches generated by different techniques*".

Proj.	SimFix	jGP	jKali	Nopol	ACS	HDR	ssFix	ELIXIR	JAID
Chart	4	0	0	1	2	-(2)	3	4	2(4)
Closure	6	0	0	0	0	-(7)	2	0	5(9)
Math	14	5	1	1	12	-(7)	10	12	1/(7)
Lang	9	0	0	3	3	-(6)	5	8	1/(5)
Time	1	0	0	0	1	-(1)	0	2	0/(0)
Total	34	5	1	5	18	13(23)	20	26	9/(25)

Unfortunately, our own experience in developing and assessing APR tools has proven that this comparison is non-trivial, and could further be largely biased due to a non-consideration of important details regarding the FL step. Indeed, recall that an APR technique cannot attempt to generate the correct patch unless the FL step can successfully identify the target buggy code locations in a program. Thus, FL accuracy across repair pipelines can impact, either by boosting or degrading, the performance of an APR system.

For example, SimFix [93] and ACS [239], although they have been developed by the same research group, are evaluated on different versions of a fault localization technique without discussing the impact of such a change in the experimental configuration. As another example bias, while most APR techniques simply integrate off-the-shelf fault localization tools in the repair pipelines, in some experiments, such as for HDRepair [124], its authors make the assumption that the buggy method is known. Unfortunately, this assumption gives an important advantage as the list of suspicious code statements is limited and likely to include the buggy statement, thus leading to overestimation of the performance.

> *Similar to the "overfitting" study, which helped to improve the assessment criteria of APR tools, our work aims at highlighting the potential biases in comparing different APR approaches without any consideration of implementation variations of the FL step.*

Overall, our investigation into the relationship between fault localization performance and APR tool performance seeks to provide answers to the following research questions (RQs):

RQ1 ***How do APR systems leverage FL techniques?*** We first investigate FL techniques used in APR systems in the literature. This reveals which FL tool and formula are integrated for each APR system. We examine implementation details of each APR system, and/or directly ask the authors of the technique to clarify FL configuration, e.g., which level of detection granularity is considered, and how many suspicious locations are considered.

RQ2 ***How many bugs from a common APR benchmark are actually localizable?*** After aggregating APR performance data reported in the literature, we note that 246 bugs (in benchmark Defects4J) have not yet been fixed by any state-of-the-art APR tool. Given that researchers scarcely discuss the reasons behind repair misses, we assess, with this research question, our intuition that FL is possibly one of the challenging steps in the repair pipeline.

RQ3 ***To what extent APR performance can vary by adapting different FL configurations?*** We implement and make publicly available `kPAR`, a straightforward fix pattern-based APR system, and record its performance under various configurations to serve as a comparable baseline for future research.

Eventually, we make the following contributions:

- We expose a hidden bias throughout the comparison of APR tools in the literature, and present more reliable performance comparisons for current state-of-the-art.
- We build and make publicly available an easy-to-configure fault localization toolkit that can be adopted in APR pipelines for Java programs.

- We provide a refined benchmark for evaluating the performance of APR systems with respect to those bugs that can actually be localized.
- We implement and make publicly available a baseline APR system with its different performance metrics for different FL configurations.

Our replication package, including `kPAR`, is available at: `https://github.com/SerVal-DTF/FL-VS-APR`

4.2 Background

We recall how fault localization is important in an APR pipeline, and describe how current APR systems are assessed.

4.2.1 Fault Localization in Automated Program Repair

In APR systems, fault localization (FL) is not only the first step but also seriously affects the performance of the systems. Given a buggy program (with its passing and failing test cases), an FL tool is leveraged during the FL step to identify the suspicious buggy code locations as described in Figure 4.1. The granularity of suspicious locations can be a file, method, or line. Ideally, the location should be both precise and accurate. If the precision is low (e.g., the granularity is broad such as file), the patch generation step needs to explore a large space of candidate patches. If the accuracy is low (e.g., the FL step provides a wrong fault location), the subsequent step generates patches for the non-faulty program entity.

Spectrum-based fault localization (SBFL, also referred to as coverage-based fault localization) [234, 241, 252] is one of the most popular FL techniques used in APR systems. This technique applies a ranking metric to detect faulty code locations by leveraging the execution traces of test cases to calculate the likelihood (based on *suspiciousness scores*) of program entities to be faulty. The ranking metric is applied to calculate suspiciousness scores for program entities (such as program statements as well as code lines [187]).

In the APR literature [91, 93, 158, 226, 239], Ochiai [4] is widely used as the ranking metric of SBFL. Many empirical studies [208, 236, 241] have indeed shown that Ochiai is one of the most effective techniques in localizing the root cause of faults in object-oriented programs. In practice, FL tools eventually report a ranked list of statements associated with the suspiciousness scores that are computed with the dedicated ranking metric.

4.2.2 APR Performance Assessment

The current practice of APR studies often evaluates the performance of APR systems based on the number of successfully fixed bugs [93, 113]. We can determine whether a generated patch is successful by counting the number of passing test cases. If a patch can pass all the given test cases (both passing and failing cases given for the buggy version), it is regarded as a successful patch.

However, the number of passing test cases may not correctly assess the effectiveness of generated patches. Even if a generated patch can pass all test cases, it might break a necessary behavior or introduce other faults, which are not covered by the given test suite [202]. Moreover, a developer may not accept the patch due to several reasons such as coding convention [104, 173]. These patches are often called **plausible patches** since it needs further investigations to check whether they are **correct patches** acceptable to developers. In the literature, *correctness* is assessed manually by comparing the generated against the developer-provided patch available in the benchmark.

Similarly, selecting a FL technique could be another issue since it can make the performance assessment biased. Our investigations will use Table 4.2 as a starting point to highlight the problem of FL bias. This table shows the number of fixed bugs out of the bugs in the Defects4J [98] benchmark, which are reported by the authors of the current state-of-the-art APR tools in the literature. The results of jGenProg, jKali and Nopol are extracted from the experimental data reported by Martinez et al. [156]. The results of other tools are collected from data reported by papers' authors in the literature.

Table 4.2: Number of bugs reported having been fixed by different APR tools. *APR systems are ordered by year of publication.*

Proj.	**jGP**	**jKali**	**jMR**	**HDR**	**Nopol**	**ACS**	**ELIXIR**	**JAID**	**ssFix**	**CapGen**	**SF**	**FM**	**LSR**	**SimFix**
Chart	0/7	0/6	1/4	0/2	1/6	2/2	4/7	2/4	3/7	4/4	6/8	5/8	3/8	4/8
Closure	0/0	0/0	0/0	0/7	0/0	0/0	0/0	5/11	2/11	0/0	3/5	5/5	0/0	6/8
Lang	0/0	0/0	0/1	2/6	3/7	3/4	8/12	1/8	5/12	5/5	3/4	2/3	8/14	9/13
Math	5/18	1/14	2/11	4/7	1/21	12/16	12/19	1/8	10/26	12/16	7/8	12/14	7/14	14/26
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0
Time	0/2	0/2	0/1	0/1	0/1	1/1	2/3	0/0	0/4	0/0	0/1	1/1	0/0	1/1
Total	5/27	1/22	3/17	6/23	5/35	18/23	26/41	9/31	20/60	21/25	19/26	25/31	19/37	34/56
P(%)	18.52	4.55	17.65	26.09	14.29	78.26	63.41	29.03	33.33	84.00	73.08	80.65	51.35	60.71

† In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool (i.e., a patch that makes the program pass all test cases). The same as other similar tables. jGP, jMR, HDR, SF, FM, and LSR denote jGenProg, jMutRepair, HDRepair, SketchFix, FixMiner, LSRepair, respectively. The same as Tables 4.4 and 4.9.

4.3 Experimental Setup

Our experiments are based on common tool-support and processes used in the literature. We clarify the experiment design in this section as the basis for understanding the implementation and the conclusions that we draw.

4.3.1 Definition of Fault Locality

Although state-of-the-art fault localization tools identify suspicious code lines, this information spans across other code entities such as methods and files, which can be sufficient for APR mutations. Thus, to compute the performance of fault localization techniques on a benchmark, we consider different granularities of fault locality at the *file*, *method* and *line* levels similar to the fault locality defined by Lucia et al. [218]:

- **File**: At this level, we consider that the faulty code is accurately localized if an FL tool reports any line from the buggy code file as suspicious.
- **Method**: At this level, we consider that the faulty code is accurately localized if any code line in the buggy method is reported by an FL tool as suspicious.
- **Line**: At this level, we consider that the faulty code is accurately localized if suspicious code lines reported by an FL tool contain any of the buggy code lines.

4.3.2 Identification of Correct Fault Locality

Our objective is to identify which reported suspicious code position is correct, following the above three levels of fault locality granularity. In practice, FL tools produce a ranked list of suspicious lines while ground truth data include several code lines as buggy lines as well. At a given granularity level, if the bug is localized (i.e., there is a match between the suspicious code line and the ground truth fault locations), we record the associated position of the correct fault locality within the ranked list of suspicious code locations. Since a bug position could span over several lines, methods, and even over

several files, the bug is considered to be correctly localized by an FL tool as long as any reported suspicious code line can match the ground truth bug locations with the corresponding granularity.

Concretely, we first use the following defintion of bug locations. The locations of a bug in a faulty program are defined as a *bug position set*: $BPos = \{bPos_1, bPos_2, \ldots, bPos_n\}$, (n >= 1), where $bPos_i$ is a tuple of ($fName$, $Methods$, $Lines$). For each location, $fName$, $Methods$, and $Lines$ are a file name, a set of methods, and a list of line numbers, respectively, of a bug location. $Methods$ could be $\emptyset$ if the bug is not located in any method in a program. This kind of bugs can be related to a Type Declaration[1] or Field Declaration[2] in Java code. *Math-12* in the Defects4J dataset is an example, which is fixed by inserting an interface `Serializable` into the type declaration[3].

We then check whether a ranked list of suspicious lines by an FL tool can identify bug locations based on the following definition. Let $SuspL = \{suspL_1, suspL_2, \ldots, suspL_m\}$ be a list of suspicious lines that are reported by an FL tool and ordered by suspiciousness scores. $suspL_i$ is a tuple of ($fName$, $lineNum$, $rIdx$), where $lineNum$ is the line number of the code in a file (i.e., $fName$) that is suspected to be the bug location, and $rIdx$ is the index (i.e., rank) of the line within $SuspL$. If a suspicious line $suspL_i$ ($i \in [1, m]$) matches any bug location ($BPos$) at a given granularity before other suspicious lines, it is considered that the FL tool successfully identifies a bug location at the given granularity. Otherwise, if there is no suspicious line matching a bug location at a given granularity, the fault is considered as non-localizable at this fault locality granularity.

4.3.3 Dataset and Automatic Testing Toolset

Our study requires execution of fault localization on a reliable dataset. In this work, we select the Defects4J [98] dataset as it includes test cases for buggy Java programs with the associated developer fixes. This dataset has furthermore been used by all recent state-of-the-art APR systems targeting Java programs. Table 4.3 summarizes statistics on the number of bugs and test cases available in the version 1.2.0[4] of Defects4J that we use in this paper.

Table 4.3: Defects4J dataset information.

Project	Chart	Closure	Lang	Math	Mockito	Time	Total
# of bugs	26	133	65	106	38	27	395
# of test cases	2,205	7,927	2,245	3,602	1,457	4,130	21,566

# of test cases are excerpted from the Defects4J paper [98] and [99].

Overall, the dataset includes 395 bugs and 22,954 test cases. To automate the execution of these test cases for each bug, we rely on the GZoltarw[5] [40] framework for automatic debugging of Java applications. GZoltar executes the test cases and produces coverage matrices providing information on which test cases passed, which failed, which statements were executed when running each test case, etc. Based on this information, FL techniques can be applied for ranking suspicious code locations which are likely to be the faulty code. For the purpose of our study, we have implemented on-top of GZoltar 41 common ranking metrics [241, 252] for fault localization. Given that Gzoltar has been used by several APR tools in the literature, we expect that our easy-to-configure fault localization toolkit will serve the research community to parameterize fault localization in an APR pipeline.

Our experiments further considered two different versions of GZoltar. The first one is the GZoltar version 0.1.1, which is already used in state-of-the-art APR systems, such as Astor [158], FixMiner [113],

[1] `http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/TypeDeclaration.html`
[2] `http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/FieldDeclaration.html`
[3] `http://program-repair.org/defects4j-dissection/#!/bug/Math/12`
[4] `https://github.com/rjust/defects4j/releases/tag/v1.2.0`
[5] `http://www.gzoltar.com/`

ACS [239], ssFix [237] and CapGen [226] among others. On the other hand, the GZoltar version 1.6.0 is used in SimFix [93] since it was recently shown to be effective [187].

4.3.4 Implementation of a Baseline APR System

Ideally, we should consider exploring an existing APR system for drawing our reference performance. Unfortunately, we face several challenges: (1) only a few research groups openly release the code or even implementation details of their APR systems; (2) repair steps are often tightly coupled together in implementation, which requires substantial engineering effort for experimental adaptation; (3) proposed approaches generally mix several contributions which are hard to isolate.

We, therefore, propose to implement and share a baseline repair system based on a state-of-the-art publication on Java program repair. We select PAR [104] for its simplicity and the straightforward replication that can be carried out on the basis of details from the relevant research report. We build `kPAR`, which leverages patterns that have been learned from the commonalities among 60,000 human-written patches. Six common patterns from the initial version of PAR has been implemented in `kPAR`. We further record the performance of `kPAR` in repair scenarios involving four different configurations of the fault localization step.

4.4 Study Results

We now provide key findings for the related questions that are investigated in this work.

4.4.1 Integration of FL Tools in APR Pipelines

To characterize how FL tools are integrated into APR pipelines, we carefully assess evaluation reports in the literature and investigate the source code (when it is available) of 14 state-of-the-art APR systems which have been evaluated on the Defects4J benchmark. Table 4.4 enumerates the studied tools along with the information collected. We focus on the testing framework that is used and its version, the FL ranking metric that is considered to compute the suspiciousness scores, the granularity of fault locality that authors focused on, and the extra information that authors use to supplement FL.

Table 4.4: Fault Localization (FL) techniques integrated into state-of-the-art APR tools.

	jGP	**jKali**	**jMR**	**HDR**	**Nopol**	**ACS**	**ELIXIR**	**JAID**	**ssFix**	**CapGen**	**SF**	**FM**	**LSR**	**SimFix**
FL testing framework	GZoltar	GZoltar	GZoltar	?	GZoltar	GZoltar	?	?	GZoltar	GZoltar	?	GZoltar	GZoltar	GZoltar
Framework version	0.1.1	0.1.1	0.1.1	?	0.0.10	0.1.1	?	?	0.1.1	0.1.1	?	0.1.1	0.1.1	1.6.0
FL ranking metric	Ochiai	Ochiai	Ochiai	?	Ochiai	Ochiai	Ochiai	?	?	Ochiai	Ochiai	Ochiai	Ochiai	Ochiai
Granularity of fault locality	line	line	line	line	line	line	line	line	line	line	line	line	method	line
Supplementary information	∅	∅	∅	Faulty method is known	∅	Predicate switching	?	?	Statements in crashed stack trace	?	?	∅	∅	Test Case Purification

* The unspecified/unconfirmed information of an APR tools is marked with '?'. If an APR tool does not use any supplementary information for FL, the corresponding table cell is marked with '∅'.

Among the 14 APR tools that are investigated, 10 leverage GZoltar as the automated testing toolset in the repair pipeline. Except for SimFix, which uses a recent version of the framework, all others use earlier versions (8 tools use version 0.1.1, while Nopol uses an even older version, i.e., 0.0.1). Thus, unless otherwise stated, the experiments in this work are performed on the widely used version 0.1.1 of GZoltar.

Eleven out of the 14 APR tools are explicitly known to rely on Ochiai for computing the suspiciousness scores in the fault localization process. This popularity of Ochiai is backed up by empirical evidence on its effectiveness to help localize faults in object-oriented programs as highlighted by several fault localization studies [186, 208, 236, 241]. A recent work by Pearson et al. [187] has even shown that Ochiai outperforms current state-of-the-art ranking metrics, or at least offers similar performance measures. In the latter part of this study, we replicate their work to ensure that our implementation of the ranking techniques is reliable. It should also be noted that although ELIXIR and SketchFix do not report the test framework that they use, they explicitly mention using Ochiai for fault localization.

With respect to the granularity of fault locality, only LSRepair [143] focuses on the method-level granularity to detect and fix bugs. Other APR systems require information on bugs at the line level to proceed with patch generation. Considering methods as the granularity of fault locations implies that such faults that are located outside methods (e.g., type declaration faults [139]) will not be addressed. However, this granularity may offer a time advantage: when several statements in a single method are reported as suspicious locations, LSRepair, unlike other APR systems, is not required to iteratively try each location for generating patch candidates. Finally, it should be noted that FL tools do not offer the same accuracy in identifying faulty locations at different granularity levels (cf. Section 4.4.2), making method level granularity appealing for limiting unnecessary trials on fault positive locations.

It is further noteworthy that four APR systems leverage supplementary information to assist the fault localization step and improve accuracy. The impact of this assistance is unfortunately never discussed when comparing performance among state-of-the-art repair approaches. Typically:

- HDRepair [124] assumes that the faulty methods are known: the fault localization step therefore focuses on ranking the lines inside the method, thus leaving out noisy statements that other APR tools are considering. This artificially reduces the probability to produce overfitting patches for HDRepair, and even increases the chance to generate a correct patch before any execution timeout.
- ssFix [237] prioritizes statements from the stack trace of crashed programs that are executed before those statements that are ranked by the FL tool.
- ACS [239] uses predicate switching [251] and refines the suspicious code locations list since the repair is focused on faulty conditional statements.
- SimFix [93] applies a test case purification [242] approach to improve the accuracy of FL step before patch generation.

Although these extra steps, which are taken to supplement FL step, could be justified intuitively, the community needs to clearly investigate their impact, in order to enable fair comparisons among the repair techniques themselves. Indeed, given that APR systems are currently compared with respect to the number of bugs that are correctly fixed, it is important that the research community reflects on what are the key contributions for explaining APR performance: for example, by counting numbers of correct patches, several programs may not be repairable by a given APR system simply because the fault is not accurately localized by the implemented FL step.

> ***RQ1***▶*State-of-the-art APR systems in the literature add some adaptations to the usual FL process to improve its accuracy. Unfortunately, researchers have eluded so far the contribution of this improvement in the overall repair performance, leading to biased comparisons.*

4.4.2 Localizability of Defects4J Bugs

In a recent work, Koyuncu et al. [113] have reported that 136 bugs in total from the Defects4J dataset have already been associated to a plausible patch that was generated by at least one APR system from the literature. Patches for 83 bugs have even been validated as correct patches by researchers. Considering this data that we complement with the performance realized by another recent APR tool,

namely LSRepair, we conclude that ∼62% (246/395) of Defects4J's bugs have never seen a plausible patch automatically generated by the state-of-the-art in APR. Although a recent empirical study [175] has suggested that current APR systems cannot repair hard and important bugs, our intuition is that there might be a more practical issue related to the localizability of Defects4J defects:

> *How many faults in the Defects4J benchmark can actually be localized by current automated fault localization tools?*

We consider the most common scenario of fault localization scenario from the APR literature: GZoltar is used for automated test execution, and Ochiai for computing suspiciousness scores. Test execution is performed with the test cases provided in the Defects4J benchmark. Table 4.5 provides quantitative details on the localizability of bugs at different levels of fault locality granularity (i.e., file, method and line). Experiments are performed with two distinct versions of GZoltar.

Table 4.5: Number of Bugs localized* with Ochiai/GZoltar.

Project	# Bugs	File		Method		Line	
		GZ_1	GZ_2	GZ_1	GZ_2	GZ_1	GZ_2
Chart	26	25	25	22	24	22	24
Closure	133	113	128	78	96	78	95
Lang	65	54	64	32	59	29	57
Math	106	101	105	92	100	91	100
Mockito	38	25	26	22	24	21	23
Time	27	26	26	22	22	22	22
Total	395	344	374	268	325	263	321

*A bug is counted as localized as long any of the faulty locations appear in the ranked list of suspicious locations reported by the FL tool. GZ_1 and GZ_2 indicate GZoltar 0.1.1 and 1.6.0, respectively. The same abbreviations are used for GZoltar versions in the following tables. The column GZ_1 of "Line" is highlighted since it is the most common configuration in APR systems.

In this experiment, we consider a bug to be localized as long as the faulty code is listed among the suspicious statements reported by this fault localization tools. Considering the most common configuration in the literature (GZoltar version 0.1.1 and "Line" granularity level), up to 132 (= 395 - 263) bugs in Defects4J are not localized. The number of bugs that are not localized decreases to 74 (= 395 - 321) when the coverage matrices are produced with GZoltar version 1.6.0. This result suggests that with GZoltar version 1.6.0, APR systems have an opportunity attempt the fix of 58 more bugs.

> ***RQ2***▶ *One third of bugs in the Defects4J dataset cannot be localized by the commonly used automated fault localization tool. Nevertheless, the recent version of GZoltar provides coverage information that helps localize more than 50 bugs, which may have never been considered in validation trials of early APR systems.*

Besides Ochiai, we have attempted to localize bugs in the Defects4J benchmark by using six other ranking metrics to compute suspiciousness scores. Table 4.6 presents the number of bugs localized by the different ranking metrics. We consider the cases where the actual fault location is reported at the Top-1 position of the suspicious code locations, and among the Top-10 positions. Results for Top-50, Top-100, Top-200 and all localized are also made available in the replication package. The results show that fault localization performance is consistent among the different ranking metrics.

Only 45 bugs can be accurately localized with Ochiai at the first suspicious line location. 140 and 214 bugs can be localized at Top-10 and Top-100 positions. Actually, many APR systems only focus on generating patches iteratively based on a part of the list of suspicious code locations. For example,

Table 4.6: Number of Bugs localized at Top-1 and Top-10.

Ranking	GZ_1			GZ_2		
Metric	**File**	**Method**	**Line**	**File**	**Method**	**Line**
Top-1 Position						
Tarantula	171	101	45	169	106	35
Ochiai	173	102	45	172	111	38
DStar2	173	102	45	175	114	40
Barinel	171	101	45	169	107	36
Opt2	175	97	39	179	115	39
Muse	170	98	40	178	118	41
Jaccard	173	102	45	171	112	39
Top-10 Position						
Tarantula	240	180	135	242	189	144
Ochiai	244	184	140	242	191	145
DStar2	245	184	139	242	190	142
Barinel	240	180	135	242	190	145
Opt2	237	168	128	239	184	135
Muse	234	169	129	239	186	140
Jaccard	245	184	139	241	188	142

for SketchFix [91], authors explicitly declare to consider only the top-50 most suspicious statements in the ranked list, while in ELIXIR [199], up to the top-200 suspicious locations are considered.

4.4.3 Impact of Effective Ranking in Fault Localization

Automated fault localization produces a ranked list of suspicious code locations that APR tools must iteratively consider for patch generation. To assess to what extent effective ranking (i.e., placing the actually faulty code locations at the top of the list), we propose to investigate the correlation between the rank of bug localization in the suspicious lists and the ability of state-of-the-art systems to be able to repair it.

Table 4.7 summarizes the list of all bugs, from the Defects4J benchmark, for which a plausible patch has been generated by one of the 14 state-of-the-art APR systems considered in this study. For each bug, we indicate the rank of the bug location within the ranked list of suspicious locations provided by the fault localization for different localization granularities. Experiments are done using the Ochiai ranking metric, but with two versions of GZoltar for computing the test coverage matrices. The raw data, including for other ranking metrics, are available in our replication package.

We propose to compute the distributions of positions across subsets of bugs for checking correlations between the localization ranking positions and the ability of APR systems to fix the bugs. Thus, we normalize bug localization positions by computing reciprocal positions based on the following formula:

$$Reciprocal_{pos}(bug_{pos}) = \begin{cases} 0, & \text{if } bug_{pos} = 0; \\ \frac{1.0}{bug_{pos}}, & otherwise. \end{cases} \tag{4.1}$$

where bug_{pos} refers to the position of the actual bug location[6] in the ranked list of suspicious locations reported by the FL step. If the bug location can be found in the higher position of the ranked list, the value of $Reciprocal_{pos}$ is closer to 1. Similarly, the value of $Reciprocal_{pos}$ trends to 0 when the bug location is at lower positions in the list of suspicious locations. This value is set to 0 when the bug cannot be localized by the FL tool (i.e., $bug_{pos} = 0$). In addition, for the purpose of our experiments, we consider three sub-classes of bugs:

[6]If several lines are concerned by the bugs, we consider the first time any of these lines appear as the bug position (cf. Section 4.2).

Table 4.7: Localization positions (i.e., rank within the suspicious list) for Defects4J bugs which have been fixed (correctly or plausibly) by corresponding APR systems.

	APR tools															GZ^1 & Ochiai			GZ^2 & Ochiai		
Bug ID	jGenProg	jKali	jMutRepair	HDRepair	Nopol	ACS	ELIXIR	JAID	ssFix	CapGen	SketchFix	FixMiner	LSRepair	SimFix	File	Method	Line	File	Method	Line
Chart-1	❍	❍	●	❍			●	(●)	●	●	●	●	●	●	1	1	28	24	24	28
Chart-3	❍				❍		❍							●	1	1	7	1	1	4
Chart-4												●	●		1	1	49	2	2	173
Chart-5	❍	❍			●				❍						1	1	7	12	66	72
Chart-6													❍		2	49	49	24	222	224
Chart-7	❍		❍						❍					●	1	2	28	1	2	75
Chart-8				❍			●			●	●				0	0	0	1	1	1
Chart-9							●	(●)	❍		●				1	1	3	1	2	14
Chart-10													❍		1	1	1	1	3	3
Chart-11							●			●	●	●	●		1	1	15	1	24	28
Chart-12												❍		❍	1	0	0	801	1092	1093
Chart-13	❍	❍			❍		❍		❍		❍	❍			1	1	17	5	35	51
Chart-14						●							❍		1	1	1	38	996	998
Chart-15	❍	❍													1	26	26	2143	8444	8445
Chart-17							❍					❍			1	2	2	11	12	12
Chart-18												❍	❍		1	1	6	1	1	3
Chart-19						●									1	1	5	37	833	833
Chart-20									●		●			●	4	0	0	62	62	62
Chart-21					❍										2	2	2	1	34	39
Chart-22													❍		1	1	1	1	53	58
Chart-24							●		●	●	●	●			1	1	2	1	1	3
Chart-25	❍	❍	❍		❍							❍			1	30	47	1325	3668	3913
Chart-26		❍	❍		❍		●				❍	●	❍		132	132	132	241	14795	15053
Closure-5								❍							8	0	0	561	0	0
Closure-7									❍						7	0	0	28	0	0
Closure-10				❍								●			3	56	120	3	67	141
Closure-12									❍						154	368	368	393	1085	1085
Closure-14				❍					●		●			●	1	2	3	2	3	3
Closure-18								●							90	1495	1527	93	2320	2377
Closure-31								(●)							215	1026	1043	214	1756	1802
Closure-33								●							2	2	289	2	2	318
Closure-38												●			1	1	34	1	1	49
Closure-40								●							9	0	0	104	0	0
Closure-42									❍						4	0	0	15	0	0
Closure-51				❍											25	25	33	41	41	50
Closure-57														●	1	2	3	1	2	7
Closure-62				❍				(●)			●	●		●	1	1	1	1	1	4
Closure-63								(●)				●		●	1	1	1	1	1	4
Closure-68									❍						2	2	2	2	2	4
Closure-70				❍				●			❍				143	0	0	264	0	0
Closure-73				❍				●			❍	●		●	1	7	10	1	1	16
Closure-79														❍	0	0	0	1	37	37
Closure-106														❍	0	0	0	3	4	4
Closure-109									❍						1	9	9	1	4	4
Closure-111									❍						1	0	0	7	0	0
Closure-115									●					●	1	1	1	8	8	8
Closure-122									❍						1	1	2	1	2	2
Closure-125								❍	❍						4	142	145	5	166	170
Closure-126				❍				(●)	❍		●				1	1	1	6	6	6
Lang-2									❍				❍		127	127	128	1	1	17
Lang-6				●			●			●	●	●			0	0	0	54	73	74
Lang-7						●									373	0	0	1	1	25
Lang-10				❍										❍	114	0	0	1	64	64
Lang-16														●	322	0	0	1	1	27
Lang-21									●				●		1	0	0	1	1	2
Lang-24						●	●	❍					●		259	618	0	1	14	64
Lang-26							●			●					21	0	0	1	112	112
Lang-27			❍						❍					●	262	269	0	1	1	56
Lang-29													●		59	59	0	1	1	0
Lang-33						●	●	●						●	14	0	0	1	1	7
Lang-35						●									53	0	0	1	1	2
Lang-38						●	(●)								104	0	0	1	3	3
Lang-39					❍	❍	❍	❍	❍					●	203	0	0	1	2	27
Lang-40													❍		1	1	1	1	1	2
Lang-41													❍	●	1	5	7	1	5	6
Lang-43				❍			●		●	●		❍		●	1	1	1	1	26	29
Lang-44					●		❍		❍					❍	1	5	20	1	1	3
Lang-45							(●)							❍	1	1	16	1	1	5
Lang-46					❍								●		1	1	3	1	1	1
Lang-48													●		1	1	2	1	1	2
Lang-50														●	1	9	15	1	8	8
Lang-51				●	❍		❍	(●)	❍		❍		●		1	1	0	1	1	0
Lang-52													●		1	1	13	1	3	25
Lang-53					❍										1	1	32	1	1	16
Lang-54													●		1	1	2	1	1	4
Lang-55				●				(●)			●		❍		1	7	9	1	6	7
Lang-57				❍			●			●		●			1	1	1	1	1	1

	APR tools															GZ^1 & Ochiai			GZ^2 & Ochiai		
Bug ID	jGenProg	jKali	jMutRepair	HDRepair	Nopol	ACS	ELIXIR	JAID	ssFix	CapGen	SketchFix	FixMiner	LSRepair	SimFix	File	Method	Line	File	Method	Line
Lang-58					●		❍		❍					●	1	2	8	1	1	20
Lang-59							●		●	●	●	●			1	1	1	1	1	6
Lang-60													❍	●	1	1	3	1	1	2
Lang-61								❍							1	14	19	1	14	21
Lang-62													❍		1	1	7	1	1	7
Lang-63									❍					❍	1	1	1	1	1	1
Math-1														❍	20	0	0	12	14	14
Math-2	❍	❍	❍				❍		❍						1	11	11	44	44	44
Math-3					●				❍						1	1	16	1	1	3
Math-4					●										1	1	1	1	3	6
Math-5	●			●	●		●		●		●			●	1	2	2	1	1	1
Math-6									❍					❍	1	2	205	1	2	177
Math-8	❍	❍							❍					❍	1	1	3	1	3	4
Math-10												●			1	1	1	5	5	17
Math-11													❍		1	9	9	27	27	29
Math-16													❍		1	1	5	1	1	5
Math-20								❍		❍				❍	1	0	0	1	0	0
Math-22				●								●			1	1	1	1	1	1
Math-25					●										0	0	0	0	0	0
Math-28	❍	❍	❍		❍				❍				❍	❍	6	6	6	13	13	13
Math-30							●			●	●		●		1	4	9	142	161	161
Math-32		❍			❍		❍	(●)							1	3	3	5	5	6
Math-33					❍		●			●	●	●	●	●	2	4	31	2	6	44
Math-34				❍			●						●		1	1	1	1	3	3
Math-35						●								●	0	0	0	1	3	5
Math-40	❍	❍	❍		❍										1	23	24	24	33	34
Math-41									●					●	1	2	6	1	37	49
Math-42					❍										1	23	26	3	57	66
Math-49	❍	❍			❍										3	4	7	5	5	7
Math-50	●	●	❍	●	●		●	(●)	●		●			●	1	1	1	1	1	1
Math-53	●			●				(●)	●	●				●	1	1	1	1	1	2
Math-57			❍		❍		●		●	●		●		●	1	1	14	1	4	4
Math-58			❍		❍		●		❍	●		●			6	6	6	223	223	223
Math-59							●		●	●	●			●	1	1	1	1	2	2
Math-60									❍						21	21	21	283	283	284
Math-61						●									0	0	0	1	1	1
Math-63							❍		❍	●			●	●	1	8	8	1	1	1
Math-65									❍	●					1	8	9	12	12	15
Math-69															1	1	2	1	48	57
Math-70	●				❍				●	●	●	●	●	●	1	1	1	1	1	1
Math-71	❍								❍					●	1	1	1	1	1	1
Math-72														❍	1	3	4	1	1	1
Math-73	●				❍	❍	❍				❍			❍	1	1	1	1	1	1
Math-75							●			●		●	●	●	1	2	2	1	1	1
Math-78	❍	❍			❍					❍					17	21	32	67	67	109
Math-79										❍		●	●	●	23	23	25	29	29	29
Math-80	❍	❍			❍		❍	(●)	●	❍			❍	❍	1	11	18	1	14	14
Math-81	❍	❍	❍		❍	❍				❍	❍	❍		❍	1	1	6	1	1	10
Math-82	❍	❍	●	❍	❍	●	●	(●)		❍	●	●		❍	2	53	60	1	76	84
Math-84	❍	❍	❍							❍		❍			1	13	30	5	18	134
Math-85	❍	❍	●		❍	●	●	(●)	❍	●	●	●		❍	1	1	36	11	11	90
Math-87					❍										1	99	100	2	109	111
Math-88				❍	❍									❍	1	1	1	1	1	1
Math-89						●							●		1	1	1	1	1	1
Math-90						●									1	1	4	1	1	3
Math-91													●		1	1	2	1	1	1
Math-93						●							❍		1	1	2	1	1	2
Math-94													●		1	1	21	1	1	21
Math-95	❍	❍								❍			❍		2	2	3	8	11	12
Math-97					❍	❍									1	1	1	1	1	1
Math-98														●	1	1	6	1	1	4
Math-99						●							❍		1	1	1	1	1	4
Math-104					❍		❍								1	0	0	1	0	0
Math-105					❍			❍							1	1	1	1	25	25
Mockito-13													●		30	30	70	74	74	135
Time-4	❍	❍					●			❍		❍			36	36	208	6	6	31
Time-7														●	48	48	51	10	10	14
Time-11	❍	❍	❍		❍		❍		❍						4	0	0	51	0	0
Time-14									❍						4	4	7	2	2	3
Time-15					●	●									1	1	115	1	1	2
Time-17									❍						5	5	5	5	5	5
Time-19				❍								●			5	449	449	104	620	620

∗ ● indicates that the bug is correctly fixed and ❍ indicates that the generated patch is plausible but not correct. (●) indicates that a correct patch is generated, but is not the first plausible patch to be generated". "0" means that the bug cannot be localized by the corresponding FL tool with the corresponding ranking metric in the corresponding granularity.

- *correctly fixed bugs*: these are bugs for which a correct patch has been provided by at least one APR tool.
- *overfitting-fixed bugs*: these are bugs for which one or more plausible patch has been generated, although none has been found to be correct.
- *unfixed bugs*: these are bugs for which no plausible patch has ever been generated by any APR system. Due to space limitation, localization data for these bugs are only available in the replication package.

Figure 4.2[7] shows the distribution of reciprocal positions for the three classes of bugs at the file, method, line granularity of fault locality. It clearly appears that correctly-fixed bugs are more accurately localized than others: i.e., their location precisions are higher in the ranked list of suspicious locations by FL tools. On the other hand, unfixed bugs tend to be those that are poorly localized: even at the file level, FL tool show low performance in localizing such bugs.

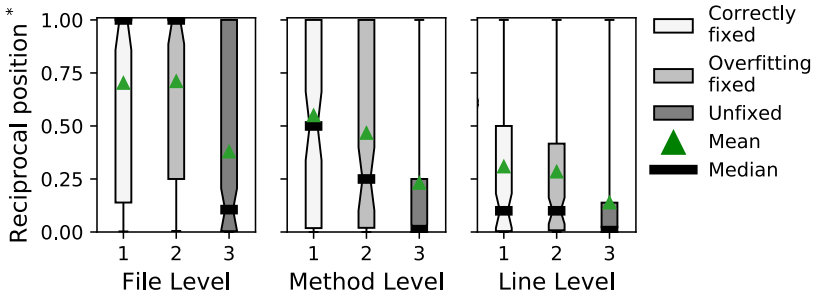


Figure 4.2: Distribution of reciprocal positions of actual bug locations among the ranked list of suspicious locations.

> ***RQ2***►*APR tools are prone to correctly fix the subset of Defects4J bugs that can be accurately localized.*

We further observe from the data in Table 4.7 that a few APR systems report patches for some bugs even though they cannot be localized (at the line level) with the configuration of Ochiai/GZoltar 0.1.1. There are various justifications to this phenomenon:

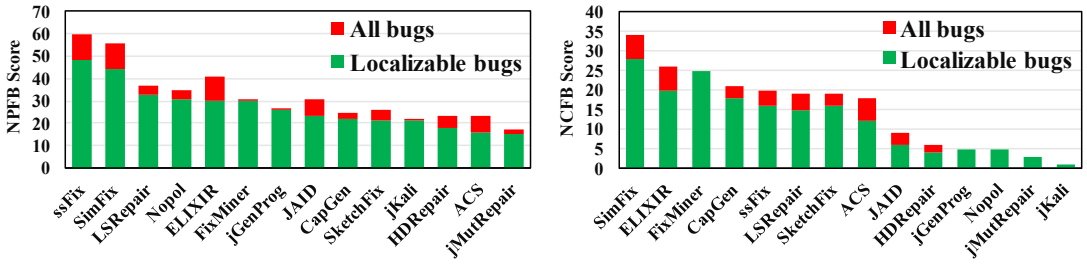
- **Improved version of the fault localization step** - *Chart-20* cannot be localized with GZoltar 0.1.1 and Ochiai, but is reported to be fixed by tools such as SimFix and ssFix. Our investigations show that SimFix has used a recent version of GZoltar (1.6.0), which is capable of localizing *Chart-20* among other bugs that were not localizable. ssFix on the other hand indeed uses GZoltar 0.1.1 but do not consider only the results of the FL tool: statements in the stack trace of crashed programs are also considered as potential fault locations.
- **Targeted localization** - HDRepair can fix *Lang-6*, which is not localized with Ochiai/GZoltar 0.1.1, because this APR system assumes that the faulty method is known, and thus directly ranks the restricted set of statements in this method. SketchFix and JAID also proceeded with such assumption on fault methods.
- **Coarse-grained repair** - LSRepair can fix four bugs which cannot be localized at the line granularity. This is due to the fact that LSRepair requires only fault localization at the method level, which is not a bias per se.
- **Non-explicit fault localization process** - ELIXIR correctly fix some bugs that are not localized under the proposed configuration. Unfortunately, besides the lack of details in their associated research reports, the source code of the tool was not made available for further investigation. *Chart-8* is another example that is not localizable by using Ochiai/GZoltar 0.1.1. This specific un-localizability problem was recently raised by Yuan and Banzhaf [249] as well as Martinez et al. [156]. Nevertheless, CapGen and ELIXIR are reported to have fixed this bug.

[7]The bug positions before being reciprocated shown in the figure are localized by GZoltar 0.1.1 with Ochiai.

> ***RQ3***▶ *APR systems do not fully disclose their fault localization tuning parameters, thus preventing reliable replication and comparisons.*

Given the bias that can be introduced by unlocalizable bugs being fixed by specific tweaking, which are not clearly outlined by the authors, we propose to count the numbers of bugs that are fixed by APR systems among those bugs that are known to be localizable. Table 4.9 thus represents an updated version of Table 4.2 where performance can be compared on the same basis. To illustrate the differences between the two comparison tables, we compute three scores: (1) **NPFB**: number of plausibly-fixed bugs, (2) **NCFB**: number of correctly-fixed bugs, and (3) **P^{3}C**: probability of plausible patch correctness.

Figures 4.3a and 4.3b illustrate the differences in respectively NPFB and NCFB scores when considering all bugs vs only localizable bugs. We note that all tools may produce some plausible patches that are plausible even for non-localizable bugs. This finding suggests that the test cases in Defects4J are insufficient since it is possible for APR systems to change non-faulty code locations and still produce patches that make the faulty program pass all test cases. On the other hand, five APR systems cannot produce any correct patches for bugs that are not localizable. ACS, ELIXIX and SimFix can correctly fix bugs that are not localized with GZoltar 0.1.1, suggesting extra impact with an improved version of the fault localization step. On the other hand, LSRepair can fix bugs that are not localized at the line level because method level fault localization is sufficient for its execution.



a # of plausibly-fixed bugs.

b # of correctly-fixed bugs.

Figure 4.3: Number of fixed bugs among all bugs vs. localizable bugs.

Finally, Table 4.8 establishes the re-ranking of APR systems in terms of the **P^{3}C** scores when focusing on localizable bugs. When focusing on localizable bugs, state-of-the-art APR systems can correctly overall fix fewer bugs than reported in the literature.

Table 4.8: Adjusted Probability of Plausible Patch Correctness.

All			**Localizable**	
P^{3}C	Rank	Tool	P^{3}C	Rank
84.0	1	CapGen	81.8	↓ 2
80.6	2	FixMiner	83.3	↑ 1
78.3	3	ACS	75.0	↓ 4
73.1	4	SketchFix	76.2	↑ 3
63.4	5	ELIXIR	66.7	5
60.7	6	SimFix	63.6	6
51.4	7	LSRepair	45.5	7
33.3	8	ssFix	33.3	8
29.0	9	JAID	26.1	9
26.1	10	HDRepair	22.2	10
18.5	11	jGenProg	19.2	↓ 12
17.6	12	jMutRepair	20.0	↑ 11
14.3	13	Nopol	16.1	13
4.5	14	jKali	4.8	14

Table 4.9: Number of localizable bugs (with GZoltar 0.1.1 and Ochiai) fixed by different APR tools.

Proj.	jGP	jKali	jMR	HDR	Nopol	ACS	ELIXIR	JAID	ssFix	CapGen	SF	FM	LSR	SimFix
Chart	0/7	0/6	1/4	0/1	1/6	2/2	3/6	2/4	2/6	3/3	4/6	5/7	3/8	3/6
Closure	0/0	0/0	0/0	0/6	0/0	0/0	0/0	3/8	2/8	0/0	3/4	5/5	0/0	6/6
Lang	0/0	0/0	0/0	0/3	3/5	0/0	3/5	0/3	2/6	3/3	2/2	2/3	4/10	5/8
Math	5/18	1/14	2/11	4/7	1/20	9/13	12/17	1/8	10/25	12/16	7/8	12/14	7/14	13/23
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0
Time	0/1	0/1	0/0	0/1	0/0	1/1	2/2	0/0	0/3	0/0	0/1	1/1	0/0	1/1
Total	5/26	1/21	3/15	4/18	5/31	12/16	20/30	6/23	16/48	18/22	16/21	25/30	15/33	28/44
Total*	5/27	1/22	3/17	6/23	5/35	18/23	26/41	9/31	20/60	21/25	19/26	25/31	19/37	34/56
P(%)	19.2	4.8	20.0	22.2	16.1	75.0	66.7	26.1	33.3	81.8	76.2	83.3	45.5	63.6
P(%)*	18.52	4.55	17.65	26.09	14.29	78.26	63.41	29.03	33.33	84.00	73.08	80.65	51.35	60.71

*Greyed-out rows are copied from Table 4.2 (i.e., numbers reported in the literature) to ease comparison with the numbers of localizable bugs that are fixed.

4.4.4 Evaluating kPAR with Specific FL Configurations

kPAR is an open-source APR system that we have built to provide a baseline for comparisons of different FL configurations. We evaluate its performance against the Defects4J benchmark with the following four different configurations of the fault localization step:

1. **Normal_FL** gives a ranked list of suspicious code locations identical as reported by a given FL tool.
2. **File_Assumption** assumes that the faulty code files are known. Suspicious code locations from Normal_FL are then filtered accordingly. In other words, locations in the known buggy files are selected and locations in other files are ignored.
3. **Method_Assumption** assumes that the faulty methods are known (the same assumption with [124]). Only locations in the known methods are selected and locations in other methods are ignored.
4. **Line_Assumption** assumes that the faulty code lines are known. No fault localization is then used.

These configurations have an order with respect to a potential size of the search space. Conceptually, the relationships between them hold $P(|Normal\_FL|) \leq P(|File\_Assumption|) \leq P(|Method\_Assumption|) \leq P(|Line\_Assumption|)$, if we consider each configuration as producing a set of suspicious locations, where $P(|*|)$ is the probability that the relevant fault locations are included in the suspicious list.

To facilitate comparison with existing repair systems, we leverage the standard GZoltar 0.1.1 and Ochiai in the following experiments. For each bug, we apply kPAR at most three hours (wall-clock time); we assume that it fails to fix a given bug if it takes more than three hours. We set this value according to the experimental setup of Astor [158]. Table 4.10 summarizes the number of bugs fixed by kPAR with the different FL configurations.

As shown in Table 4.10, kPAR can fix its maximum number of bugs when the accurate fault locations are provided (i.e., with *Line_Assumption*). With this assumption, kPAR can correctly fix 36 bugs in Defects4J, a record performance in the literature (not accounting for the bias in the fault localization step).

Table 4.10: # of Bugs fixed by kPAR.

FL Configuration	Chart (C)	Closure (Cl)	Lang (L)	Math (M)	Mockito (Moc)	Time (T)	Total
Normal_FL	3/10	5/9	1/8	7/18	1/2	1/2	18/49
File_Assumption	4/7	6/13	1/8	7/15	2/2	2/3	22/48
Method_Assumption	4/6	7/16	1/7	7/15	2/2	2/3	23/49
Line_Assumption	7/8	11/16	4/9	9/16	2/2	3/4	36/55

Figure 4.4 further details which bugs are fixed in the different configurations. First, we note that all bugs fixed with a given localization configuration are also fixed by any of the relatively more accurate fault localization configurations. Thus, with the *File_Assumption* configuration, kPAR can fix not only all bugs that were already fixed with the *Normal_FL* configuration but also can now fix four more bugs. By examining the case of those four bugs, we figure out that, in the case of two bugs (i.e., *Cl-4* and *T-19*), the faulty locations were ranked very low in *Normal_FL*, leading to an execution stop due to timeout. For the remaining two bugs (i.e., *C-26* and *Moc-29*), however, in *Normal_FL*, kPAR is led to consider first some irrelevant suspicious statements that made kPAR to generate plausible patches that are not correct. Given that the repair process stops when a plausible patch is produced, there is no opportunity with *Normal_FL* to try all suspicious statements.

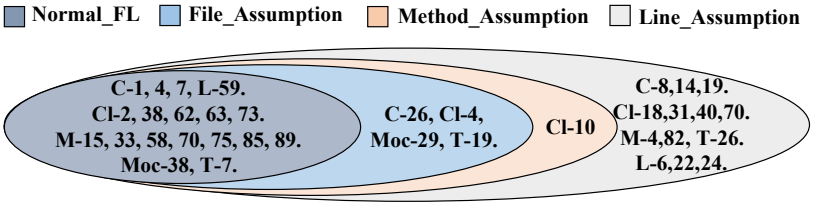


Figure 4.4: Bugs correctly fixed by kPAR with four configurations.

When filtering the set of suspicious locations with *Method_Assumption*, kPAR can fix one more bug (i.e., *Cl-10*), which could not be fixed by other two less confined FL configurations (i.e., *Normal_FL* and *File_Assumption*) before the time-out. Finally, when assuming that the fault locations are known (i.e., *Line_Assumption*), kPAR can further fix 13 bugs. These could not be fixed in other three less confined configurations. Among the 13 bugs, seven bugs (i.e., *C-8*, *Cl-40*, *Cl-70*, *L-6*, *L-22*, *L-24*, and *T-26*) are not even localizable using Ochiai/GZoltar 0.1.1; two bugs (i.e., *Cl-18* and *Cl-70*) are not fixed due to execution timeout; one bug (i.e., *M-82*) is not fixed in other three configurations since the proposed plausible patches are incorrect; three bugs (i.e., *C-14*, *C-19* and *M-4*) are partially fixed in the other three FL configurations since they have several faulty code fragments.

> ***RQ3***▶*Accuracy of fault localization has a direct and substantial impact on the performance of APR repair pipelines.*

We examine the bug *Chart-14* from the Defects4J dataset, which involves four fault code locations[8] If we regard those as four sub-bugs, each one can be correctly detected and fixed by kPAR using the *Normal_FL* configuration. However, if the exact faulty statements are unknown, kPAR (as current APR tools) iteratively mutates suspicious statements one by one in the ranked list. Even if any one of them is correctly fixed, there are still three failed tests, meaning that the generated patch (even if was a correct patch) will not even be considered as a plausible patch.

Considering a patch that partially passes some previously-failing test cases (without introducing new failing test cases) may nevertheless be harmful as it can prevent the generation of a fully correct patch. For example, *Chart-4* is a single-location bug that makes 22 test cases fail[9] Before generating the correct patch, kPAR had generated patches that made the program pass subsets of the test cases.

Other bugs, such as *Math-72*, on the other hand include multiple faulty locations that fail on the same test case. Although kPAR could generate correct patches for each faulty location, the fix process of kPAR prevents a full fix of this bug. If the test suite can be automatically augmented with differentiating test cases for each fault location, an APR system would be more successful as suggested in [245].

[8] http://program-repair.org/defects4j-dissection/#!/bug/Chart/14
[9] http://program-repair.org/defects4j-dissection/#!/bug/Chart/4

> ***RQ3***▶*APR researchers must investigate the trade-off between fixing multi-locations bugs versus bugs failing multiple test cases.*

4.5 Discussion

Our study draws a number of conclusions that we reformulate into guidelines for assessing APR systems. We further enumerate the associated threats to validity before discussing the related work.

4.5.1 APR Assessment Guidelines

- **Full disclosure of FL parameters.** Given that many APR systems do not release their source code, it is important that the experimental reports clearly indicate the protocol used for fault localization. Preferably, authors should strive to assess their performance under a standard and replicable configuration of fault localization.
- **Qualification of APR performance.** To ensure that novel approaches to APR are indeed improving over the state-of-the-art, authors must qualify the performance gain brought by the different ingredients of their approaches.
- **Patch generation step vs Repair pipeline.** There are two distinct directions of repair benchmarking that APR researchers should consider. In the first, a novel contribution to the patch generation problem must be assessed directly by assuming a perfect fault localization. In the second, for ensuring realistic assessment w.r.t. industry adoption, the full pipeline should be tested with no assumptions on fault localization accuracy.
- **Sensitivity of search space.** Given that fault localization produces a ranked list of suspicious locations, it is essential to characterize whether exact locations are strictly necessary for the APR approach to generate the correct patches. For example, an APR system may not focus only on a suspected location but on the context around this location. APR approaches may also use heuristics to curate the FL results.

4.5.2 Threats to Validity

A threat to external validity of our study is that we focus on the localizability of bugs in the Defects4J dataset, which target Java code and may not include sufficient test cases. This threat is however limited given that we investigate performance differences. Threats to internal validation include the use of a single automatic testing framework, namely GZoltar (Not all APR systems in the literature use it to localize faults.), and the selection of the 14 state-of-the-start APR systems. These threats are mitigated by the fact that we ensured that these choices are common among the APR literature.

4.5.3 Related Work

The software development practice is increasingly accepting generated patches [111]. Recently, various directions in the literature have explored to contribute to the advancement of automated program repair [4, 92, 97, 187, 190, 218, 234]. We now discuss the few related studies that attempt to investigate fault localization in relationship with APR.

Qi et al. [193] have evaluated the effectiveness of FL tools by using APR performance as a proxy. Their study proposed the NCP score[10] as the effectiveness metric. The results show that a specific FL ranking metric (Jaccard [44]) outperforms other metrics. Our study, however, reveals that the common technique used in APR is still Ochiai. Yang et al. [244] studied the usage of FL techniques

[10]NCP: number of candidate patches generated before a valid patch is found [193].

in APR systems by investigating two different algorithms of how to interpret the results of FL techniques: (1) the rank-first algorithm based on suspiciousness rankings of statements, and (2) the suspiciousness-first algorithm based on suspiciousness scores of statements. They ran Nopol [240] to compare NCP scores, repair time, and patch diversity of the two algorithms. The study concludes that the suspiciousness-first algorithm is more effective for APR systems. The above two studies, however, do not consider whether the patches generated by APR tools are correct or plausible while our study examines how FL techniques affect the quality of patches generated by APR systems.

The literature also includes work on the impact of the fault space, although it does not clarify how FL tools affect the performance of APR systems. Wen et al. [225] investigated the influence of the fault space on the success of finding correct patches by the APR tool. The fault space is defined as a ranked list of suspicious entities in a program. They examined both plausible and correct patches. However, their work is limited to evaluating a single APR tool, GenProg [128] and a single FL technique, Ochiai [3] while our study evaluates and compares 14 different APR systems. Our study further considers the exact location of faults and its correlation with the possibility of generating plausible patches. Finally, our study targets unveiling biases among APR systems.

To the best of our knowledge, our work is the first time to systematically study to what extent FL techniques impact the performance of automated program repair pipeline.

4.6 Summary

The momentum of research in automated program repair is a decisive opportunity for the software engineering research community. Every couple of months, a new APR system is proposed in a race to fix more bugs automatically. Unfortunately, validation of these systems often have only the dataset in common: important parameters such as the fault localization settings are eluded, leading to biased comparisons among the state-of-the-art. Our investigations into these biases call for new guidelines for assessing and reporting on the performance of APR systems. In particular, our replication package includes a full dissection of the Defects4J benchmark in terms of fault localization, a light-weight and tuneable fault localization toolkit, as well as a baseline Java APR system to encourage fair and reproducible experiments.

5 Fix Pattern Mining for FindBugs Violations

In this chapter, we proposed a deep learning based approach to mining fix patterns for static analysis violations. We first collect and track a large number of fixed violations across revisions of software. To automatically identify patterns in violations and their fixes, we propose an approach that utilizes convolutional neural networks to learn features and clustering to regroup similar instances. We then evaluate the usefulness of the identified fix patterns by applying them to unfixed violations. The results show that developers will accept and merge a majority (69/116) of fixes generated from the inferred fix patterns. It is also noteworthy that the yielded patterns are applicable to four real bugs in the Defects4J major benchmark for software testing and automated repair.

This chapter is based on the work published in the following research paper:

- Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018

Contents

5.1 Overview

Modern software projects widely use static code analysis tools to assess software quality and identify potential defects. Several commercial [70,183,211] and open-source [69,72,90,154] tools are integrated into many software projects, including operating system development projects [111]. For example, Java-based projects often adopt `FindBugs` [90] or `PMD` [69] while C projects use `Splint` [72], `cppcheck` [154], or `Clang Static Analyzer` [145], while Linux driver code are systematically assessed with a battery of static analyzers such as Sparse and the LDV toolkit. Developers may benefit from the tools before running a program in real environments even though those tools do not guarantee that all identified defects are real bugs [25].

Static analysis can detect several types of defects such as security vulnerabilities, performance issues, and bad programming practices (so-called code smells) [58]. Recent studies denote those defects as **static analysis violations** [220] or **alerts** [83]. In the remainder of this paper, we simply refer to them as *violations*. Figure 5.1 shows a violation instance, detected by `FindBugs`, which is a violation tagged `BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS`, as it does not comply with the programming rule that the implementation of method `equals(Object obj)` should not make any assumption about the type of its `obj` argument[1]. As later addressed by developers via a patch represented in Figure 5.2, the method should return `false` if `obj` is not of the same type as the object being compared. In this case, when the type of `obj` argument is not the type of `ModuleWrapper`, a `java.lang.ClassCastException` should be thrown.

```
1    public boolean equals(Object obj) {
2        // Violation Type: BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS
3        return getModule().equals(((ModuleWrapper) obj).getModule());
4    }
```

Figure 5.1: Example of a detected violation, taken from *PopulateRepositoryMojo.java* file at revision `bdf3fe` in project `nbm-maven-plugin`[2].

```
1    public boolean equals(Object obj) {
2 -      return getModule().equals(((ModuleWrapper) obj).getModule());
3 +      return obj instanceof ModuleWrapper && getModule().equals(((ModuleWrap
             per) obj).getModule());
4    }
```

Figure 5.2: Example of fixing violation, taken from Commit `0fd11c` of project `nbm-maven-plugin`.

Static analysis tools are widely adopted in the industry (e.g., FindBugs has more than 270K downloads[3]). Developers always make source code changes as in the example of Figure 5.2 to fix the violations spotted by static analysis tools. We investigate in this study two research questions regarding (**RQ1**) *how are the violations resolved when developers make changes?* Based on this question, for each violation type, we can derive fix patterns that may help summarize common violation (or real bug) resolutions and may be applied to fixing similar unfixed violations. (**RQ2**) *can fix patterns help systematize the resolution of similar violations?* This question may shed some light on the effectiveness of common fix patterns when applying them to potential defects.

To answer the two questions, we investigate violation fixing changes collected from 730 open source Java projects. Although the approach is generic to any static bug detection tool, we focus on a single tool, namely `FindBugs`, applying it to every revision of each project. We thus identify violations in each revision and further enumerate cases where a pair of consecutive revisions involve the resolution of a violation through source code change (i.e., the violation is found in revision r_1 and is absent from

[1] `http://findbugs.sourceforge.net/bugDescriptions.html#BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS`
[2] `https://github.com/mojohaus/nbm-maven-plugin`
[3] `http://findbugs.sourceforge.net/users.html`

r_2 after a code change can be mapped to the violation location): we refer to such recorded changes as *violation fixing changes*.

After collecting violation fixing changes from a large number of projects using an AST differencing tool [60], we mine developer fix patterns for static analysis violations. The approach encodes a fixing change into a vector space using Word2Vec [71], extracts discriminating features using Convolutional Neural Networks (CNNs) [176] and regroups similar changes into a cluster using *X-means* clustering algorithm [188]. We then evaluate the suitability of the mined fix patterns by applying them to 1) a subset of unfixed violations in our subjects, to 2) a subset of faults in Defects4J [98] and to 3) a subset of violations in 10 open source Java projects.

Overall, this study makes the following contributions:

1. **Violation fix pattern mining:** we propose an approach to infer common fix patterns of violations leveraging CNNs and *X-means* clustering algorithm. Such patterns can be leveraged in subsequent research directions such as automated refactoring tools (for complying with project rules as done by checkpatch[4][5] in the Linux kernel development), or automated program repair (by providing fix ingredients to existing tools such as PAR [104]).

   > *Mined fix patterns can be leveraged to help developers rapidly and systematically address high-priority cases of static violations. In our experiments, we showed that 40% of a sample set of 500 unfixed violations could be immediately addressed with the inferred fix patterns.*

2. **Pattern-based violation patching:** we apply the fix patterns to unfixed violations and actual bugs in real-world programs. Our experiments demonstrate the potential of the approach to infer patterns that are effective which shows the potential of automated patch generation based on the fix patterns.

   > *Developers are ready to accept fixes generated based on mined fix patterns. Indeed out of 113 generated patches, 69 were merged in 10 open source projects. It is noteworthy that since static analysis can uncover important bugs, mined patterns can be leveraged for automated repair. Out of the 14 real-bugs in the Defects4J benchmark which can be detected with `FindBugs`, our mined fix patterns are immediately applicable to produce correct fixes for 4 bugs.*

The remainder of this paper is organized as follows. We propose our study method in Section 5.2, describing the process of violation tracking, and the approach for mining code patterns based on CNNs. Section 5.3 presents the study results in response to the research questions. Limitations of our study are outlined in Section 5.4. Section 5.5 surveys related work. We conclude the paper in Section 5.6 with discussions of future work.

5.2 Methodology

Our study aims at uncovering common fix patterns related to developers' fixes for static analysis violations. As shown in Figure 5.3, our study method unfolds in four steps: (1) applying a static analysis tool to collecting violations from programs, (2) tracking violations across the history of program revisions, (3) identifying fixed violations, (4) mining common fix patterns in each class of fixed violations. We describe in details these steps as well as the techniques employed.

5.2.1 Collecting Violations

To collect violations from a program, we apply a static analysis tool to every revision of the associated project's source code. Given the resource-intensive nature of this process, we focus in this study on

[4] `http://tuxdiary.com/2015/03/22/check-kernel-code-checkpatch`
[5] `https://github.com/spotify/linux/blob/master/scripts/checkpatch.pl`

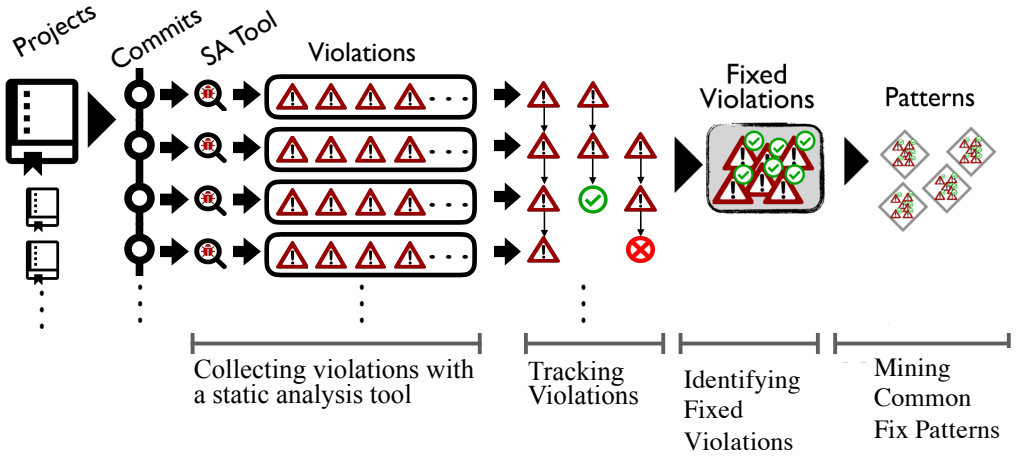


Figure 5.3: Overview of our study method.

the `FindBugs` [201] tool, although our method is applicable to other static analysis tools such as Facebook Infer[6], Google ErrorProne[7], etc. We use the most sensitive option to detect all types of violations defined in `FindBugs` violation descriptions[8]. For each individual violation instance, we record, as a six-tuple value, all information on the violation type, the enclosing program entity (e.g., project, class or method), the commit id, the file path, and the location (i.e., start and end line numbers) where the violation is detected. Figure 5.4 shows an example of a violation record in the collected dataset.

```
<ViolationInstance>
    <ViolationType>NP_NULL_ON_SOME_PATH</ViolationType>
    <ProjectName>GWASpi-GWASpi</ProjectName>
    <CommitVersionID>b0ed41</CommitVersionID>
    <FilePath>src/main/java/org/gwaspi/gui/reports/Report_AnalysisPanel.java
    </FilePath>
    <StartLineNumber>89</StartLineNumber>
    <EndLineNumber>89</EndLineNumber>
</ViolationInstance>
```

Figure 5.4: Example record of a single-line violation of type `NP_NULL_ON_SOME_PATH` found in *ReportAnalysisPanel.java* file within Commit `b0ed41` in `GWASpi`[9] project.

Since `FindBugs` requires Java bytecode rather than source code, and given that violations must be tracked across all revisions in a project, it is necessary to automate the compilation process. In this study, we accept projects that support the *Apache Maven* [18] build automation management tool. We apply maven build command (i.e., '*mvn package install*') to compiling each revision in 2,014 projects that we have collected. Eventually, we were able to successfully build 730 automatically.

5.2.2 Tracking Violations

Violation tracking consists in identifying identical violation instances between consecutive revisions: after applying a static analysis tool to a specific revision of a project, one can obtain a set of violations. In the next version, another set of violations can be produced by the tool. If there is any change in the next revision, new violations can be introduced and existing ones may disappear. In many cases however, code changes can move violation positions, making this process a non-trivial task.

[6] http://fbinfer.com/
[7] https://errorprone.info/
[8] `http://findbugs.sourceforge.net/bugDescriptions.html`
[9] https://github.com/GWASpi/GWASpi

Static analysis tools often report violations with line numbers in source code files. Even when a commit modifies other lines in different source file than the location of a violation, it might be unable to use line numbers for matching identical violation pairs between two consecutive revisions. Yet, if the tracking is not precise, the identification of fixed violations may suffer from many false positives and negatives (i.e., identifying unfixed ones as fixed ones or vice versa). Thus, to match potential identical violations between revisions, our study follows the method proposed by Avgustinov et al. [21]. This method has three different violation matching heuristics when a file containing violations is changed. The first heuristic is (1) *location-based matching*: if a (potential) matching pair of violations is in code change diffs[10], it compares the offset of the corresponding violations in the code change diffs. If the difference of the offset is equal to or lower than 3, we regard the matching pair as an identical violation. When a matching pair is located in two different code snapshots, we use (2) *snippet-based matching*: if two text strings of the code snapshots (corresponding to the same type of violations in two revisions) are identical, we can match those violations. When the two previous heuristics are not successful, our study applies (3) *hash-based matching*, which is useful when a file containing a violation is moved or renamed. This matching heuristic first computes the hash value of adjacent tokens of a violation. It then compares the hash values between two revisions. We refer the reader to more details on the heuristics in [21].

There have been several other techniques developed to do this task. For example, Spacco et al. [206] proposed a fuzzy matcher. It can match violations in different source locations between revisions even when a source code file has been moved by package renaming. Other studies [79, 82] also provide violation matching heuristics based on software change histories. However, these are not precise enough to be automatically applied to a large number of violations in a long history of revisions [21].

5.2.3 Identifying Fixed Violations

Once violation tracking is completed, we can figure out the resolution of an individual violation. Violation resolution can result in three different outcomes. (1) A violation can disappear due to deleting a file or a method enclosing the violation. (2) A violation exists at the latest revision after tracking (even some code is changed), which indicates that the violation has not been fixed so far. (3) A violation can be resolved by changing specific lines (including code line deletion) of source code. The literature refer to the first and second outcomes as *unactionable violations* [79, 82, 84] or *false positives* [206, 246, 248] while the third one is called *actionable violations* or *true positives*. In this study we inspect violation tracking results, focusing on the third outcome (which yield the set of **fixed violations**).

Starting from the earliest revision where a violation is seen, we follow subsequent revisions until a later revision has no matching violation (i.e., the violation is resolved by removal of the file/method or the code has been changed). If the violation location in the source code is in a diff pair, we classify it as a *fixed violation*. Otherwise, it is an *unfixed violation*.

5.2.4 Mining Common Fix Patterns

Our goal in this step is to summarize how a violation is resolved by developers. To achieve this goal, we collect violation fixing changes and proceed to identify their common fix patterns. The approach of mining common fix patterns is similar to that of mining common code patterns. The differences lie in the data collection and tokenization process. Before describing our approach of mining common fix patterns, we formalize the definitions of patch and fix pattern.

[10] A "code change diff" consists of two code snapshots. One snapshot represents the code fragment that will be affected by a code change, and another one represents the code fragment after it has been affected by the code change.

5.2.4.1 Preliminaries

A patch represents a modification carried on a program source code to repair the program which was brought to an erroneous state at runtime. A patch thus captures some knowledge on modification behavior, and similar patches may be associated with similar behavioral changes.

Definition 1 ***Patch (P):*** *A patch is a pair of source code fragments, one representing a buggy version and another as its updated (i.e., bug-fixing) version. In the traditional GNU diff representation of patches, the buggy version is represented by lines starting with* `-`*, while the fixed version is represented by lines starting with* `+`*. A patch is formalized as:*

$$P = (Frag_b, Frag_f) \tag{5.1}$$

where $Frag_b$ and $Frag_f$ are fragments of buggy/fixing code, respectively; both are a set of text lines. Either of the two sets can be an empty set but cannot be empty simultaneously. If $Frag_b = \emptyset$, the patch purely adds a new line(s) to fix a bug. On the other hand, the patch only removes a line(s) if $Frag_f = \emptyset$. Otherwise (i.e., both sets are not empty), the patch replaces at least one line.

Figure 5.5 shows an example of a patch which fixes a bug of converting a `String List` into a `String Array`. $Frag_b$ is the line that starts with `-` while $Frag_f$ is the lines that start with `+`. By analyzing the differences between the buggy code and the fixing code of the patch in Figure 5.5, the patch can be manually summarized as an abstract representation shown in Figure 5.6 which could be used to address similar bugs. Abstract representation indicates that specific identifiers and types are abstracted from concrete representation.

```
1 DiffEntry of a patch:
2 @@ -1246,1 +1246,1 @@
3 - return (String[]) list.toArray(new String[0]);
4 + return (String[]) list.toArray(new String[list.size()]);
```

Figure 5.5: Example of a patch taken from *FilenameUtils.java* file within Commit `09a6cb` in project `commons-io`[11].

```
1 Abstract representation of a patch:
2 @
3 var: a list variable.
4 T: the parameterized type of a list.
5 @
6 - (T[]) var.toArray(new T[0]);
7 + (T[]) var.toArray(new T[var.size()])
```

Figure 5.6: Example of an abstract representation of the patch in Figure 5.5.

Abstract patch representations can be formally defined as *fix patterns*. Coccinelle [135] and its semantic patches provide a metavariable example of how fix patterns can be leveraged to systematically apply common patches, e.g., to address collateral evolution due to API changes [184]. Manually summarizing fix patterns from patches is however time-consuming. Thus, we are investigating an automated approach of mining fix patterns. To that end, we first provide a formal definition of a fix pattern.

Definition 2 ***Fix Pattern (FP):*** *A fix pattern is a pair of a code context extracted from a buggy code block and a set of change operations, which can be applied to a given buggy code block to generate fixing code. This can be formalized as:*

$$FP = (Ctx, CO) \tag{5.2}$$

[11] `https://commons.apache.org/proper/commons-io/`

where *Ctx* represents the code context that is an abstract representation of the buggy code block (See Definition 3). *CO* is a set of change operations (See Definition 5) to be applied to modifying the buggy code block.

Definition 3 ***Code Context (Ctx):*** *A code context is a three-element tuple, which is extracted from a fined-grained AST subtree associated to a code block, i.e.,*

$$Ctx = (Sce, Sce_p, cctx) \tag{5.3}$$

where *Sce* is a source code entity and Sce_p is the parent entity of *Sce* (with $Sce_p = \emptyset$ when *Sce* is a root entity). *cctx* is a list of code contexts that are the children of *Ctx*. When *Sce* is a leaf node entity, $cctx = \emptyset$.

Definition 4 ***Source Code Entity (Sce):*** *A source code entity (hereafter entity) is a pair of type and identifier, which denotes a node in an Abstract Syntax Tree (AST) representation, i.e.,*

$$Sce = (Type, Identifier) \tag{5.4}$$

where *Type* is an AST node type and *Identifier* is a textual representation (i.e., raw token) of an AST node, respectively.

Definition 5 ***Change Operation (O):*** *A change operation is a three-value tuple which contains a change action, a source code entity and a set of sub change operations. This can be formalized as:*

$$O = (Action, Sce, CO) \tag{5.5}$$

where *Action* is an element of an action set (i.e., {`UPD, DEL, INS, MOV`}) working on the entity (*Sce*). `UPD` is an *update* action which means updating the target entity, `DEL` is a *delete* action which denotes deleting the target entity, `INS` is an *insert* action which represents inserting a new entity, and `MOV` is a *move* action which indicates moving the target entity. *CO* is a set of sub change operations working on the sub entities of the current action's entity. When an operation acts on a leaf node entity, $CO = \emptyset$.

For example, Figure 5.7 shows the set of change operations of the patch in Figure 5.5. o_1 is the change operation working on the root entity `ReturnStatement`. `UPD` is the *Action*, (`ReturnStatement`, `return`) is the root entity being acted, and o_2 is the sub change operation acting on the sub entity `CastExpression` of the root entity. It is the same as others. o_6, o_8, and o_9 are the change operations working on leaf node entities. So that, the sets of their sub change operations are null.

***A Set of Change Operations*:**
o1. (UPD, (ReturnStatement, return), [
o2. (UPD, (CastExpression, (String[])), [
o3. (UPD, (MethodInvocation, list.toArray), [
o4. (UPD, (Method, toArray), [
o5. (UPD, (ArrayCreation, String), [
o6. (DEL, (ArraySize, NumberLiteral), ∅)
o7. (INS, (MethodInvocation, list.size), [
o8. (INS, (Variable, list), ∅),
o9. (INS, (Method, size), ∅)])])])])])])

Figure 5.7: A set of change operations of the patch in Figure 5.5.

5.2.4.2 Fix Pattern Mining Process

Figure 5.5 shows a concrete patch that can only be used to fix related specific bugs as it limits the syntax and semantic structure of the buggy code. The statement is limited to be a `Return Statement` and the parameterized type of the `List` and the `Array` is also limited to `String`. Additionally, the variable name `list` can also interfere with the matching between this patch and similar bugs. However, the abstract patch in Figure 5.6 abstracts the aforementioned interferon, which can be matched with various mutations of the bug converting a `List` into an `Array`. Hence, it is necessary to mine common patch patterns from massive and various patches for specific bugs.

Our conjecture is that *common fix patterns* can be mined from large change sets. Exposed bugs are indeed generally not new and common fix patterns may be an immediate and appropriate way to address them automatically. For example, when discussing the deluge of buggy mobile software, Andy Chou, a co-designer of the Coverity bug finding tool, reported that, based on his experience, the found bugs are nothing new and are "actually well-known and well-understood in the development community - the same *use after free* and *buffer overflow* defects we have seen for decades" [25]. In this vein, we design an approach to mine common fix patterns for static analysis violations by extracting changes that represent developers' manual corrections. Figure 5.8 illustrates our process for mining common fix patterns.

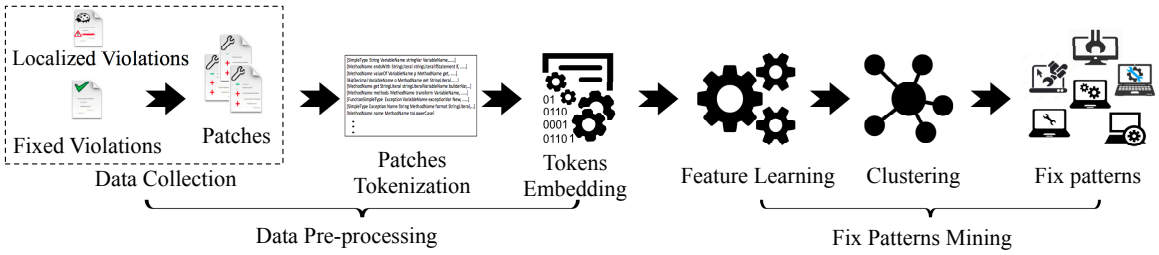


Figure 5.8: Overview of our fix patterns mining method.

Data Preprocessing As defined in Definition 2, a fix pattern contains a set of change operations, which can be inferred by comparing the buggy and fixed versions of source code files. In our study, code changes of a patch are described as a set of change operations in the form of Abstract Syntax Tree (AST) differences (i.e., AST diffs). In contrast with GNU diffs, which represent code changes as a pure text-based line-by-line edit script, AST diffs provide a hierarchical representation of the changes applied to the different code entities at different levels (statements, expressions, and elements). We leverage the open source GumTree [60] tool to extract and describe change operations implemented in patches. GumTree, and its associated source code, is publicly available, allowing for replication and improvement, and is built on top of the Eclipse Java model[12].

All patches are tokenized into textual vectors by traversing their AST-level diff tree with the deep-first search algorithm and extracting the action string (e.g., `UPD`), the entity type (e.g., *ReturnStatement*) and the entity identifier (e.g., *return*) as tokens of a change action (e.g., *UPD ReturnStatement return*). A given patch is thus represented as a list of such tokens.

Token Embedding with Word2Vec Widely adopted deep learning techniques require numeric vectors with the same size as the format of input data. Tokens embedding is performed with Word2Vec [169] which can yield a numeric vector for each unique token. Word2Vec[13] [169] is a two-layer neural network, whose main purpose is to embed words, i.e., convert each word into a numeric vector. Eventually, a patch is then embedded as a two-dimensional numeric vector (i.e., a vector of the vectors embedding the tokens). Since token vectors may have different sizes throughout violations, the corresponding numeric vectors must be padded to comply with deep learning algorithms

[12] `http://www.vogella.com/tutorials/EclipseJDT/article.html`
[13] `https://code.google.com/archive/p/word2vec/`

requirements. We follow the workaround tested by Wang et al. [222] and append 0 to all vectors to make all vector sizes consistent with the size of the longest vector.

Numerical representations of tokens can be fed to deep learning neural networks or simply queried to identify relationships among words. For example, relationships among words can be computed by measuring cosine similarity of vectors, given that Word2Vec strives to regroup similar words together in the vector space. Lack of similarity is expressed as a 90-degree angle, while complete similarity of 1 is expressed as a 0-degree angle. For example, in our experiment, 'true' and 'false' are `boolean` literal in Java. There is a cosine similarity of 0.9433768 between 'true' and 'false', the highest similarity between 'true' and any other token.

The left side of Figure 5.9 shows how a patch is vectorized. The $n \times k$ represents a two-dimensional numeric vector of an embedded and vectorized patch, where n is the number of rows and denotes the size of the token vector of a patch. A row represents a numeric vector of an embedded token. k is the number of columns and denotes the size of a one-dimensional numeric vector of an embedded token. The last two rows represent the appended 0 to make all numeric vector sizes consistent.

Fix Patterns Mining Patches can be considered as a special kind of natural language text, which programmers leverage daily to request and communicate changes in their community. Currently available patch tools only perform directly the specified operations (e.g., remove and add lines for GNU diff) so far without the interpretation of what the changes are about. Although all patches can be parsed and converted into two-dimensional numeric vectors, it is still challenging to mine fix patterns given that noisy change information (e.g., specific changes) can interfere with identifying similar patches. Thus, our method is designed to effectively learn discriminating features of patches for mining fix patterns. We leverage CNNs to perform deep learning of patch features with preprocessed patches, and *X-means* clustering algorithm to automatically cluster similar patches together with learned features. Finally, we manually label each cluster with fix patterns of violations abstracted from clustered patches to show fix patterns clearly.

Feature Learning with CNNs Figure 5.9 shows the CNNs architecture for learning patch features. The input is two-dimensional numeric vectors of preprocessed patches. The alternating local-connected convolutional and subsampling layers are used to capture the local features of patches. The dense layer compresses all local features captured by former layers. We select the output of the dense layer as the learned patch features to cluster patches. Note that our approach uses CNNs to extract features of code change actions of patches, in contrast to normal supervised learning applications that classify labels with training process to show patterns clearly.

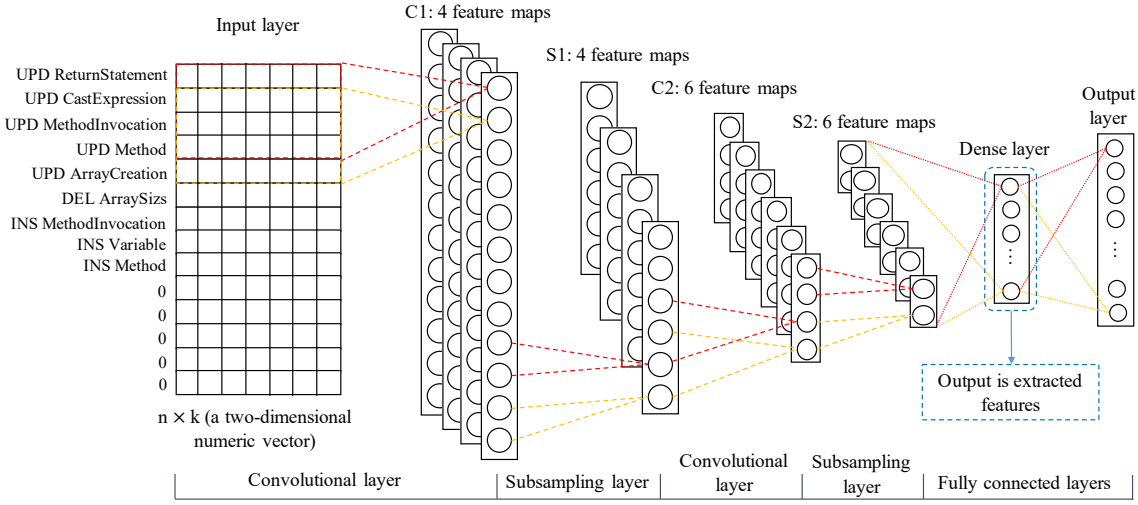


Figure 5.9: CNN architecture for extracting clustering features. *C1* is the first convolutional layer, and *C2* is the second one. *S1* is the first subsampling layer, and *S2* is the second one. The output of dense layer is considered as extracted features of code change actions and will be used to do clustering.

Patches Clustering and Patterns Labelling With learned features of patches, cluster patches with *X-means* clustering algorithm. In this study, we use Weka's implementation [230] of *X-means* to cluster violations. Finally, we manually label each cluster with identified fix patterns of patches from clustered similar code changes of patches to show fix patterns clearly.

5.3 Empirical Study

5.3.1 Datasets

We consider project subjects based on a curated database of `Github.com` provided through `GHTorrent` [65]. We select projects satisfying three constraining criteria: (1) a project has, at least, 500[14] commits, (2) its main language is Java, and (3) it is unique, i.e., not a fork of another project. As a result, 2,014 projects are initially collected. We then filter out projects which are not automatically built with *Apache Maven*. Subsequently, for each project, we execute `FindBugs` on the compiled[15] code of its revisions (i.e., committed version). If a project has at least two revisions in which `FindBugs` can successfully identify violations, we apply the tracking procedure described in Section 5.2.2 to collecting data.

Table 5.1 shows the number of projects and violations used in this study. There are 730 projects with 291,615 commits. After applying our violation tracking method presented in Section 5.2.2 to these violations, as a result, 88,927 violation patches are collected. These violation patches are associated with 400 types defined by `FindBugs`.

Table 5.1: Subjects used in this study.

# Projects	730
# Commits	291,615
# Violation patches	88,927
# Violations types	400

5.3.2 Fix Patterns Mining

We first investigate our ultimate research question on *how are the violations resolved if fixed?* (**RQ1**). To that end, we first dissect the violation fixing changes and propose to cluster relevant fixes to infer common fix patterns following the CNN-based approach described in Section 5.2.4.

We curate our dataset of 88,927 violation fixing changes by filtering out changes related to:

- 4,682 violations localized in test files. Indeed, we focus on mining patterns related to developer changes on actual program code.
- 7,010 violations whose fix do not involve a modification in the violation location file. This constraint, which excludes cases where long data flow may require a fixing change in other files, is dictated by our automation strategy for computing the AST edit script, which is simplified by focusing on the violation location file.
- 7,121 violations where the associated fix changes are not local to the method body of the violation.
- 25,464 violations where the fixing changes are applied relatively far away from the violation location. We consider that the corresponding AST edit script matches if the change actions are performed within ±3 lines of the violation location. This constraint conservatively helps to further remove false positive cases of violations which are actually not fixed but are identified as fixed violations due to limitations in violation tracking.

[14] A minimum number of commits is necessary to collect a sufficient number of violations, which will be used for violation tracking.

[15] `FindBugs` runs on compiled bytecode (cf. Section 5.2.1).

- 9,060 violations whose code or whose fix code contain a large number of tokens. In previous works, Herzig et al. [85] and Kawrykow et al. [101] have found that large source code change hunks generally address feature additions, refactoring needs, etc., rather than bug fixes. Pan et al. [185] also showed that large bug fix hunk pairs do not contain meaningful bug fix patterns, and most bug fix hunk pairs (91-96%) are small ones. Ignoring large hunk pairs has minimal impact on analysis. Consequently, we use the threshold (i.e., 40, presented in Figure 5.10) of violation tokens to select fixed violations.

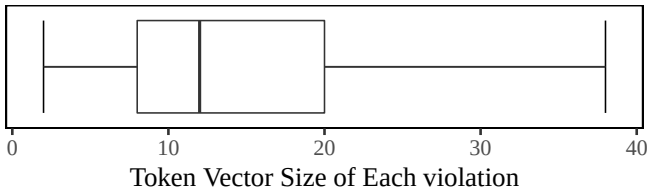


Figure 5.10: Sizes' distribution of all violation token vectors.

Overall, our fix pattern mining approach is applied to 35,590 violation fixing changes, which are associated with 288 violation types. Following the methodology described in Section 5.2.4, violations are represented with numeric vectors using Word2Vec with the following parameters (Size of vector = 300; Window size = 4; Min word frequency = 1)

Feature extraction is then implemented based on CNNs whose parameters are listed in Table 5.2. The literature has consistently reported that effective models for Word2Vec and deep learning applications require well-tuned parameters [46, 71, 115, 170, 250]. In this study, all parameters of the two models are tuned through a visualizing network training UI[16] provided by DeepLearning4J.

Table 5.2: Parameters setting of CNNs.

Parameters	Values
# nodes in hidden layers	1000
learning rate	1e-3
Optimization algorithm	stochastic gradient descent
pooling type	max pool
activation (output layer)	softmax
activation (other layers)	leakrelu
loss function	mean squared logarithmic error

Finally, Weka's [230] implementation of *X-means* clustering algorithm uses the extracted features to find similar code change actions pf patches for each violation type. Parameter settings for the clustering are enumerated in Table 5.3.

Table 5.3: Parameters setting of X-means.

Parameters	Values
Distance Function	Euclidean Distance
KD Tree	true
# max iterations	1000
# max K-means	500
# max K-means of children	500
# seed	500
# max clusters	500
# min clusters	1

In this study, once a cluster of similar changes, for a given violation type, are found, we can automatically mine the patterns based on the AST diffs. Although approaches such as the computation of longest common subsequence of repair actions could be used to mine fix patterns, we observe

[16] `https://deeplearning4j.org/visualization`

that they do not always produce semantically meaningful patterns. Thus, we consider a naive but empirically effective approach of inferring fix patterns by considering the most recurring AST edit script in a given cluster, i.e., the code change that occurs identically the most. Finally, labels to each change pattern are assigned manually after a careful assessment of the pattern relevance. For the experiments, we focus on the top-50 fixed violation types for the mining of fix patterns. Table 5.4 summarizes 10 example cases of violation types with details, in natural language, on the fix patterns.

Figure 5.11 presents an inferred pattern in terms of AST edit script for violation type `RCN_REDUNDANT_NULLCHECK _OF_NONNULL_VALUE` described in Table 5.4. For AST-level representation of patterns of other violations, we refer the reader to the replication package.

Table 5.4: Common fix pattern examples of fixed violations.

Violation Type	Fix Pattern(s)
DM_CONVERT_CASE	ADD a rule of Locale.ENGLISH into toLowerCase()/toUpperCase().
RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	① Delete the null check expression. ② Delete the null check IfStatement.
BC_UNCONFIRMED_CAST	① Delete the violated statement, ② Delete the cast type, ③ Replace CastExpression with a null value.
MS_SHOULD_BE_FINAL	Add a "final" modifier.
RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	① Add an IfStatement to check the return value of violated source code. ② Replace violated expression with a new method invocation.
DM_NUMBER_CTOR	Replace the number constructor with a static number.valueOf() method.
SBSC_USE_STRINGBUFFER_CONCATENATION	Replace the String type with the StringBuilder, and replace plus operator of StringVarialbe with the append method of StringBuilder.
DM_BOXED_PRIMITIVE_FOR_PARSING	Replace Number.valueOf() with Number.parseXXX() method.
PZLA_PREFER_ZERO_LENGTH_ARRAYS	① Delete the buggy statement, ② Replace the null value with an empty array.
ES_COMPARING_STRINGS_WITH_EQ	Replace the "==" or "!=" InfixExpression with a equals() method invocation.

```
1  DiffEntry of a patch:
2  -   if (dictionaryObject!=null && !onlyEmptyFields){
3  +   if (!onlyEmptyFields) {
4        signatures.put(new COSObjectKey(dict), (COSDictionary)dict.getObject());
5      }
6
7  Repair actions parsed by GumTree at AST level:
8  UPD IfStatement@@if(dictionaryObject!=null && !onlyEmptyFields)
9  --UPD InfixExpression@@dictionaryObject!=null && !onlyEmptyFields
10 ---DEL InfixExpression@@dictionaryObject!=null
11 -----DEL VariableName@@dictionaryObject
12 -----DEL Operator@@!=
13 -----DEL NullLiteral@@null
14 ---DEL Operator@@&&
15
16 Inferred Fix Pattern:
17 UPD IfStatement@@if(Null-Check-Expression Operator Other-Expression)}@)
18 --UPD InfixExpression@@Null-Check-Expression Operator Other-Expression
19 ---DEL Null-Check-Expression
20 ---DEL Operator
```

Figure 5.11: Example of a fix pattern for `RCN_REDUNDANT_NULL CHECK_OF_NONNULL_VALUE` violation inferred from a violation fix instance taken from `commit a41eb9` in project `apache-pdfbox`[17].

Overall, the pattern presented in AST edit script format, which should be translated into fix changes to "*delete the null check expression*" requires some code context to be concretized. When the `var`[18] `!= null` expression is the *null-checking* conditional expression of an `IfStatement`, the concrete patch must delete the violated expression. Similarly, when the `exp == null` expression is the condition expression of an `IfStatement`, the patch also removes the *null-checking* expression. When `exp == null` or `exp != null` expression is one of the condition expressions of an `IfStatement`, the patch is deleting the violated expression. This example shows the complexity of automatically generating

[17] `https://github.com/apache/pdfbox`
[18] `var` represents any variable being checked.

patches from abstract fix patterns, an entire research direction which is left for future work. For now, we generate the patches manually based on the mined fix patterns.

> *Our proposed fix pattern mining approach can effectively cluster similar changes of fixing violations together. And the fix pattern mining protocol is applicable to derive meaningful patterns.*

Listing 5.1 enumerates 12 violation types for which our mining approach could not yield patterns, given that the number of samples per cluster was small, or that within a cluster we could not find strictly redundant change actions sequences. Our observations of such cases revealed the following causes of failure in fix pattern mining:

- violations can be fixed by adding completely new node types. For example, one fix pattern of `RV_RETURN_VALUE_ IGNORED_BAD_PRACTICE` violations is replacing the violated expression with a method invocation which encapsulates the detailed source code changes.
- violations can occur on specific source code fragments from which it is even difficult to mine patterns. Fixes for such violations generally do not share commonalities.
- violations can have fix changes applied in separate region than the violation code location. Since we did not consider such cases for the mining, we systematically miss bottom-7 violation types of Listing 5.1 which are in this case.
- violations can be associated to a `String` literal. For example, we observe that the fixing changes of `VA_FORMAT_STRING_USES_NEWLINE` violations are replacing "\n" with "%n" within strings. Unfortunately, our AST nodes are focused on compilable code tokens, and thus changes in `String` literal are ignored to guarantee sufficient abstraction from concrete patches.

```
1  UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR
2  SF_SWITCH_NO_DEFAULT
3  UWF_UNWRITTEN_FIELD
4  IS2_INCONSISTENT_SYNC
5  VA_FORMAT_STRING_USES_NEWLINE
6  SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING
7  OBL_UNSATISFIED_OBLIGATION
8  OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE
9  OS_OPEN_STREAM
10 OS_OPEN_STREAM_EXCEPTION_PATH
11 ODR_OPEN_DATABASE_RESOURCE
12 NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE
```

Listing 5.1: Violation types failed to be identified fix pattern

5.3.3 Usage and Effectiveness of Fix Patterns

We finally investigate whether *fix patterns can actually help resolve violations in practice?* (**RQ2**). To that end, we consider the following sub-questions:

- RQ2-1: Can fix patterns be applied to automate the management of some *unfixed* violations?
- RQ2-2: Can fix patterns be leveraged as ingredients for automated repair of *buggy* programs?
- RQ2-3: Can fix patterns be effective in systematizing the resolution of `FindBugs` violations *in the wild*?

We recall that our work automates the generation of fix patterns. Patch generation is out of scope, and thus will be performed manually (based on the mined fix patterns), taking into account the code context.

5.3.3.1 Resolving Unfixed Violations

We apply fix patterns to a subset of unfixed violations in our subjects following the process illustrated in Figure 5.12. For a given unfix violation, we search for the top-k[19] most suitable fix patterns to generate patches. To that end, we consider cosine similarity between the violation code features vector (built with CNNs in Section 5.2.4) and the features vector of the centroid fixed violation in the cluster associated to each fix pattern.

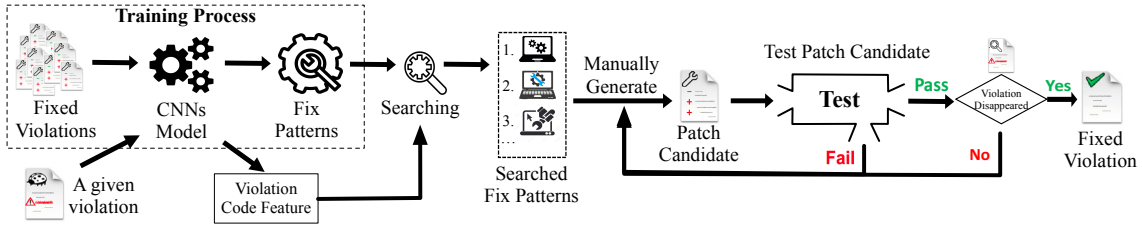


Figure 5.12: Overview of fixing similar violations with fix patterns.

A fix pattern is regarded as a true positive fix pattern for an unfixed violation, if a patch candidate derived from this pattern is addressing the violation. We check this by ensuring that the resulting program after applying the patch candidates passes compilation and all tests, FindBugs no longer raises a warning at this location, and manual checking by the authors has not revealed any inappropriate change of semantics in program behaviour.

Test Data We collect a subset of unfixed violations in the top-50 fixed violation types as the testing data of this experiment to evaluate the effectiveness of fixed patterns. For each violation type, at most 10 unfixed violation instances, which are the most similar to the centroids of the corresponding fixed violations clusters, are selected as the evaluation subjects.

Results Table 5.5 presents summary statistics on unfixed violations resolved by our mined fix patterns. Overall, among the selected 500 unfixed violations in the test data, 127 (25.4%) are fixed by the most similar matched fix patterns (i.e., top-1), 188 (37.6%) are fixed by a pattern among the top-5, and 203 (40.6%) are fixed within the top-10. The matched positive fix patterns mainly cluster on top-5 fix pattern candidates, which are a few less than the top-10 range. This suggests that enlarging the search space of fix pattern candidates cannot effectively find positive fix patterns for more target violations.

Among the 203 fixed unfixed-violations, only 3 of them are fixed by matched fix patterns collected across violation types. We observe that `DM_NUMBER_CTOR` and `DM_FP_NUMBER_CTOR` have similar fix patterns. We use the fix patterns of `DM_FP_NUMBER_CTOR` to match fix pattern candidates for `DM_NUMBER_CTOR` violations. The fix patterns of `DM_FP_NUMBER_CTOR` can fix the `DM_NUMBER_CTOR` violations, and vice versa.

> *Almost half of the unfixed violations in a sampled dataset can be systematically resolved with mined fix patterns from similar violations fixed by developers. 1 out of 4 of these unfixed violations are immediately and successfully fixed by the first selected fix pattern.*

We note that fix patterns for 23 violation types are effective in resolving any of the related unfixed violations. There are various reasons for this situation, notably related to the specificity of some violation types and code, the imprecision in `FindBugs` violation report, or the lack of patterns.

[19] $k = 10$ in our experiments

Table 5.5: Unfixed-violations resolved by fix patterns.

Violation types	Top 1	Top 5	Top 10	Total
RI_REDUNDANT_INTERFACES	10	10	10	10
SE_NO_SERIALVERSIONID	10	10	10	10
UPM_UNCALLED_PRIVATE_METHOD	10	10	10	10
DM_NUMBER_CTOR	9	10	10	10
DM_FP_NUMBER_CTOR	9	10	10	10
DM_BOXED_PRIMITIVE_FOR_PARSING	8	9	10	10
DM_CONVERT_CASE	7	9	10	10
MS_SHOULD_BE_FINAL	7	9	9	10
PZLA_PREFER_ZERO_LENGTH_ARRAYS	7	7	8	10
RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE	6	8	8	10
RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	6	7	8	10
SBSC_USE_STRINGBUFFER_CONCATENATION	4	10	10	10
MS_PKGPROTECT	4	9	9	10
EI_EXPOSE_REP2	4	4	5	10
DM_DEFAULT_ENCODING	4	5	5	10
WMI_WRONG_MAP_ITERATOR	3	7	9	10
UC_USELESS_CONDITION	3	6	6	10
ES_COMPARING_STRINGS_WITH_EQ	2	8	10	10
RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	3	4	4	10
SIC_INNER_SHOULD_BE_STATIC_ANON	3	3	3	10
UCF_USELESS_CONTROL_FLOW	2	9	10	10
BC_UNCONFIRMED_CAST_OF_RETURN_VALUE	2	4	4	10
DLS_DEAD_LOCAL_STORE	2	3	4	10
NP_NULL_ON_SOME_PATH	1	5	7	10
BC_UNCONFIRMED_CAST	1	1	1	10
UC_USELESS_OBJECT	0	8	8	10
NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE	0	3	5	10
VA_FORMAT_STRING_USES_NEWLINE	0	0	0	10
UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	0	0	0	10
DE_MIGHT_IGNORE	0	0	0	10
EI_EXPOSE_REP	0	0	0	10
IS2_INCONSISTENT_SYNC	0	0	0	10
NM_METHOD_NAMING_CONVENTION	0	0	0	10
NP_LOAD_OF_KNOWN_NULL_VALUE	0	0	0	10
NP_NONNULL_RETURN_VIOLATION	0	0	0	10
NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE	0	0	0	10
OBL_UNSATISFIED_OBLIGATION	0	0	0	10
OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE	0	0	0	10
ODR_OPEN_DATABASE_RESOURCE	0	0	0	10
OS_OPEN_STREAM	0	0	0	10
OS_OPEN_STREAM_EXCEPTION_PATH	0	0	0	10
REC_CATCH_EXCEPTION	0	0	0	10
RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT	0	0	0	10
SF_SWITCH_NO_DEFAULT	0	0	0	10
SIC_INNER_SHOULD_BE_STATIC	0	0	0	10
SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING	0	0	0	10
ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	0	0	0	10
URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD	0	0	0	10
URF_UNREAD_FIELD	0	0	0	10
UWF_UNWRITTEN_FIELD	0	0	0	10
Total	127 (25.4%)	188 (37.6%)	203 (40.6%)	500

Identified fix patterns are applied to fixing a subset of unfixed violations in our subjects.

5.3.3.2 Fixing Real Bugs

We attempt to apply fix patterns to relevant bugs documented in the Defects4J [98] collection of real-world defects in Java programs. This dataset is used in studies of program repair [124,156,239].

Test Data We run `FindBugs` on the 395 buggy versions of the 6 Java projects used to establish Defects4J. As a result, it turns out that 14 bugs can be detected as static analysis violations detectable `FindBugs`. This is a reasonable number since most of the bugs in Defects4J are functional bugs which fail under specific test cases rather than programming rule violations.

For each relevant bug, we consider the fix patterns associated to their violation types, and manually

generate the patches. When the generated patch candidate can (1) pass the failed test cases of the corresponding bug and (2) `FindBugs` cannot identify any violation at the same position, then the matched fix pattern is regarded as a positive fix pattern for this bug.

Results Table 5.6 shows the results of this experiment. 4 out of the 14 bugs are fixed with the mined fix patterns and the generated patches by fix patterns are semantically equivalent to the patches provided by developers for these bugs. The violations of 2 bugs are indeed eliminated by fix patterns, but the generated patches lead to new bugs (in terms of test suite pass). There are 2 bugs that can be matched with fix patterns, but more context information was necessary to fix them. For example, bug `Lang23` is identified as a `EQ_DOESNT_OVERRIDE_EQUALS` violation and matched with a fix pattern: overriding the `equals(Obj o)` method. It is difficult to generate a patch of the bug with this fix pattern without knowing the property values of the object being compared. The remaining 6 (out of 14 bugs) occurred on specific code, which is challenging to match plausible fix patterns for them without any context.

Table 5.6: Fixed bugs in Defects4J with fix patterns.

Classification	# bugs
Fixed bugs	4
Violations are removed but generates new bugs	2
Need more contexts	2
Failed to match plausible fix patterns	6
Total	14

Static analysis violations can represent real bugs that make programs fail functional test cases. Our mined fix patterns can contribute to automating the fix of such bugs as experimented on the Defects4J dataset.

5.3.3.3 Systematically Fixing `FindBugs` Violations in the Wild

We conduct a live study to evaluate the effectiveness of fix patterns to systematize the resolutions of violations in open source projects. We consider 10 open source Java projects collected from `Github.com` on 30th September 2017 and presented in Table 5.7. `FindBugs` is then run on compiled versions of the associated programs to localize static analysis violations.

Table 5.7: Ten open source Java projects.

Project Name	# files	# lines of code
json-simple	12	2,505
commons-io	117	28,541
commons-lang	148	77,577
commons-math	841	186,425
ant	859	219,506
cassandra	1,625	216,192
mahout	1,145	222,345
aries	1,570	216,646
poi	4,562	894,514
camel	8,119	1,079,671

Test data We focus on violation instances in the top-50 fixed violation types (presented in Table 5.5) are randomly selected as our evaluating data. For each violation, patches are generated manually in a similar process than the previous experiments: a patch must lead to a program that compiles, passes the test cases, and the previous violation location should not be flagged by `FindBugs` anymore. For each of such patch, we create a pull request and submit the patch to the project developers.

Results Overall, we managed to push 116 patches to the developers of the 10 projects (cf. Table 5.8). 30 patches have been ignored while 15 have been rejected. Nevertheless, 2 patches have been improved by developers and 67 have been immediately merged. 1 of our pull requests to the `json-simple` project was not merged, but an identical patch has been applied later by the developers to fix the violation. Finally, the last patch (out the 116) has not been applied yet, but was attached to the issue tracking system, probably for later replacement.

Table 5.8: Results of live study.

Project Name	# Patches				
	pushed	ignored	rejected	improved	merged
json-simple	2	1	0	0	0
commons-io	2	0	2	0	0
commons-lang	7	5	1	1	0
commons-math	6	6	0	0	0
ant	16	2	4	1	9
cassandra	9	9	0	0	0
mahout	3	2	0	0	0
aries	5	5	0	0	0
poi	44	0	0	0	44
camel	22	0	8	0	14
Total[†]	116	30	15	2	67

[†]One patch of `json-simple` is the same as a patch of the same violation which has been fixed by its developer in another version. One patch of `mahout` is attached to its bug report system but has not yet been merged.

Table 5.9 presents the distribution of delays before acceptance for the 69 accepted (merged + improved) patches. 67% of the patches are accepted within 1 day, while 97% (67% +30%) are accepted within 2 days. Only 2 patches took a longer time to get accepted. We note that this acceptance delay is much shorter than the median distributions of the three kinds of patches submitted for the Linux kernel [111].

Table 5.9: Delays until acceptance.

Delay	less than 1 day	1 to 2 days	17 days
Number of Patches	46 (67%)	21 (30%)	2 (3%)

*Acceptance indicates one of improved or merged patches.

As summarized in Table 5.10, we note that 21 accepted patches were verified by at least two developers. Although 48 accepted patches were verified by only one developer, we argue that this does not bias the results: first, the common source code patterns of these accepted fixed violation types are consistent with the descriptions documented by `FindBugs`; second, the matched fix patterns are likely acceptable by developers since the patterns are common in fixing violations as mined in the revision histories of real-world projects.

Table 5.10: Verification of accepted patches.

Verified by	1 developer	2 developers	3 developers
Number of Patches	48	19	2

Our mined fix patterns are effective to fix violations in the wild. Furthermore, the generated patches are eventually quickly accepted by developers.

The live study further yields a number of insights related to static analysis violations.

Insight 1. Well-maintained projects are not prone to violating commonly-addressed violation types. We note that 8 violation types (presented in Listing 5.2) do not appear at the current revisions of the selected 10 projects. Type `RI_REDUNDANT_INTERFACES` occurs only one time in `json-simple` project. This finding suggests that violation recurrences may be time-varying, so that, there is a time-variant issue of violation recurrences in revision histories of software projects, which may help to prioritize violations. It is included in our future work.

```
SIC_INNER_SHOULD_BE_STATIC
NM_METHOD_NAMING_CONVENTION
SIC_INNER_SHOULD_BE_STATIC_ANON
NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE
NP_NONNULL_RETURN_VIOLATION
UPM_UNCALLED_PRIVATE_METHOD
ODR_OPEN_DATABASE_RESOURCE
SE_NO_SERIALVERSIONID
```

Listing 5.2: Violation types not seen in the selected 10 projects.

Insight 2. Developers can write positive patches to fix bugs existing in their projects based on the fix patterns inferred with our method. For example, the developers of `commons-lang` [20] project fixed a bug[21] reported as a `DM_CONVERT_CASE` violation by `FindBugs` by improving the patch that was proposed using our method (cf. Figure 5.13). Our method cannot generate the patch they wanted because there is no fix pattern that is related to adding a rule of `Locale.ROOT` in our dataset, so that there might be a limitation of existing patches in revision histories.

```
The patch generated by fix patterns:
- final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase());
+ final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase(Locale.ENGLISH));

The patch improved by developers:
- final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase());
+ final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase(Locale.ROOT));
```

Figure 5.13: Example of an improved patch in real project.

Insight 3. Developers will not accept plausible patches that appear unnecessary even if those are likely to be useful. For example, Figure 5.14 shows a rejected patch that adds an `instanceof` test to the implementation of `equals(Object obj)`. The developers want to accept this patch at the first glimpse, but they reject to change the source code after reading the context of these violations since the implementation of `equals(Object obj)` belongs to an inner static class which is only used in a generic type that will not compare against other Object types.

```
Rejected Patch:
- return Arrays.equals(keys,((MultipartKey)obj).keys);
+ return obj instanceof MultipartKey && Arrays.equals(keys, ((MultipartKey)
    obj).keys);
```

Figure 5.14: Example of a rejected patch in real projects.

Insight 4. Some violations fixed based on the mined fix patterns may break the backward compatibility of other applications, leading developers to reject patches for such violations. For example, Figure 5.15 shows a rejected patch of a `MS_SHOULD_BE_FINAL` violation in *Path.java* file of the `ant` project, which breaks the backward compatibility of `systemClasspath` in *InternalAntRunner* class [22] of Eclipse project.

[20] https://github.com/apache/commons-lang
[21] https://garygregory.wordpress.com/2015/11/03/java-lowercase-conversion-turkey/
[22] https://github.com/eclipse/eclipse.platform/blob/R4_6_maintenance/ant/org.eclipse.ant.core/src_ant/org/eclipse/ant/internal/core/ant/InternalAntRunner.java#L1484

```
1 Rejected Patch breaks the backward compatibility:
2 -    public static Path systemClasspath =
3 +    public static final Path systemClasspath =
4                         new Path(null,System.getProperty("java.class.path"));
5
6 Code @Line 1484 in InternalAntRunner.java in Eclipse project:
7 org.apache.tools.ant.types.Path.systemClasspath = systemClasspath;
```

Figure 5.15: Example of a rejected patch breaking the backward compatibility.

Insight 5. Some violation types have low impact. For example, `PZLA_PREFER_ZERO_LENGTH_ARRAYS` refers to the `FindBugs`' rule that an array-return method should consider returning a zero-length array rather than null. Its fix pattern is replacing the null reference with a corresponding zero-length array. Developers ignored or rejected patches for this type of violations because they do have null-check to prevent these violations. If there is no null-check for these violations, the invocations of these methods would be identified as `NP_NULL_ON_SOME_PATH` violations. Thus, `PZLA_ PREFER_ZERO_LENGTH_ARRAYS` might not be useful in practice.

Insight 6. Some fix patterns make programs fail to compile. For example, the common fix pattern of `RV_RETURN_ VALUE_IGNORED_BAD_PRACTICE` violations is adding an `if` statement to check the return `boolean` value of the violated source code. We note that return values of some violated source code of this violation type is not `boolean` type. Copying the change behavior of the fix pattern directly to this kind of violations will lead to compilation errors.

Insight 7. Some fix patterns make programs fail to checkstyle. Figure 5.16 presents an example of a patch generated by our method for a `MS_SHOULD_BE_FINAL` violation in *XmlConverter.java* file of `camel` [23] project, which makes the project fail to checkstyle.

```
1 A Patch makes program fail to checkstyle:
2 -  public static String defaultCharset =
3 +  public static final String defaultCharset =
4    ObjectHelper.getSystemProperty(Exchange.DEFAULT_CHARSET_PROPERTY, "UTF-8");
```

Figure 5.16: Example of a patch making program fail to checkstyle.

Insight 8. Some fix patterns of some violations are controversial. For example, the fix patterns of `DM_NUMBER_CTOR` violations are replacing the Number constructor with static Number `valueOf` method. It has been found that changing `new Integer()` to `Integer.valueOf()` and changing `Integer.valueOf()` to `new Integer()` were reverted repeatedly. Some developers find that `new Integer()` outperforms `Integer.valueOf()`, and some other developers find that `Integer.valueOf()` outperforms `new Integer()`. Additionally, some developers report that `Double.doubleToLongBits()` could be more efficient than `new Double()` and `Double.valueOf()` when comparing two double values with `equals()` method. We infer that the `DM_NUMBER_CTOR` or `DM_FP_NUMBER_CTOR` violations should be identified and revised based on the specific function, otherwise, developers may be prone to ignoring these kinds of violations.

5.4 Discussion

5.4.1 Threats to Validity

A major threat to external validity of our study is the focus on `FindBugs` as the static analysis tool, with specific violation types and names. Fortunately, the code problems described by

[23] `https://github.com/apache/camel`

FindBugs violations are similar to the violations described by other static analysis tools. For example, NP_NULL_ON_SOME_PATH violations in FindBugs, Null dereference violations in Facebook Infer, and ThrowNull violations in Google ErrorProne are about the same issue: A NULL pointer is dereferenced and will lead to a NullPointerException when the code is executed. With the fix pattern of NP_NULL_ON_SOME_PATH of FindBugs mined in this study, we fixed 9 out of 10 different cases (each is from a distinct project in our subjects) of Null dereference violations detected by Facebook Infer and 8 out of 10 different cases of ThrowNull violations detected by Google ErrorProne, respectively. It shows the potential generalizability of the inferred fix patterns. We acknowledge, however, that there are some differences between FindBugs violations and other static analysis violations. Another threat to external validity of our study is that the fix patterns of violations are mined from open-source projects. Our findings might not applicable to industry projects that could have specific policies related to code quality.

Threats to internal validity include the limitations of the underlying tools used in this study (i.e., FindBugs and GumTree). GumTree may produce unfeasible edit scripts. To reduce this threat, we have added extra labels into GumTree. FindBugs may produce some violations with inaccurate positions. To reduce this threat, we re-locate and re-visit the violated source code with the bug descriptions of some violation types by FindBugs. FindBugs may yield high false positives. In order to reduce this threat, we focus on the common fixed violations in this study since common fixed violations are really concerned by developers. If the common fixed violations were addressed by common fix patterns, the common fixed violations are highly possible to be true positives and the common fix patterns are highly possible to be effective resolutions. These threats could be further reduced by developing more advanced tools.

Threats to internal validity also involve limitations in our method. Violation tracking may produce false positive fixed violations. We combine the commit DiffEntry and diffs parsed by GumTree to reduce this threat. Irrelevant code contexts can interfere with patterns mining. For example, one statement contains complex expressions, which may lead to a high number of irrelevant tokens. If this kind of violations were not filtered out in this study, it would increase the interference of noise. To reduce this validity, our study should be replicated in future work by extracting and analyzing the key violated source code with relevant code contexts identified using system dependency graphs. In this study, we also find that some violations are replaced by method invocations which encapsulate the detailed source code changes of fixing the corresponding violations. The method we proposed extracts source code changes from source code changing positions of violations. It is challenging to extract source code changes from these kinds of fixed violations. In order to reduce this validity, we are planning to integrate static analysis technique into our method to get more detailed source code changes.

5.4.2 Insights on Unfixed Violations

Given the high proportion of violations that were found to remain unfixed in software projects, we investigate the potential reasons for this situation. We note that the reasons are not mainly due to the violation code characteristics. Instead, we can enumerate other implicit reasons based on the observation of statistical data as well as the comments received during our live study to fix violations in ten open source projects.

- Actually, many developers do not use FindBugs as part of their development tool chain. For example, we found that only 36% of projects in our study include a commit mentioning FindBugs. Also, interestingly, in the cases of projects where we found that only 2% (1,944/88,927) of fixed FindBugs violations explicitly refer to the FindBugs tool in commit messages.
- Our interaction with developers helped us confirm that developers do not consider most FindBugs violations as being severe enough to deserve attention in their development process.

- Some violations identified by `FindBugs` might be controversial because we find that some fix patterns of some violations are controversial (cf. Insight 8 in Section 5.3.3.3).
- Finally, with our live study, we note that some developers may be willing to fix violations if they had in hand some fix patterns. Unfortunately, `FindBugs` only reports the violations, and does not provide in many cases any hint on how to deal with them. Our work is towards filling this gap systematically based on harvested knowledge from developer fixes.

5.5 Related work

5.5.1 Change Pattern Mining

Empirical Studies on Change Patterns Common change patterns are useful for various purposes. Pan et al. [185] explored common bug fix patterns in Java programs to understand how developers change programs to fix a bug. Their fix patterns are, however, in a high-level schema (e.g. "If-related: Addition of Post-condition Check (IF-APTC)"). Based on the insight, PAR [104] leveraged common pre-defined fix patterns for automated program repair, that only contain six fix patterns which can only be used to fix a small number of bugs. Martinez and Monperrus further investigated repair models that can be utilized in program fixing while Zhong and Su [254] conducted a large-scale study on bug fixing changes in open source projects. Tan et al. [213] analyzed anti-patterns that may interfere with the process of automated program repair. However, all of them studied code changes at the statement level, which is not as fine-grained as our work that extracts fine-grained code changes with an extended version of GumTree [60].

Pattern Mining for Code Change SYDIT [167] and Lase [168] generate code changes to other code snippets with the extracted edit scripts from examples in the same application. RASE [166] focuses on refactoring code clones with Lase edit scripts [168]. FixMeUp [204] extracts and applies access control templates to protect sensitive operations. Their objectives are not to address issues caused by faulty code in program, such as the static analysis bugs studied in this study. REFAZER implements an algorithm for learning syntactic program transformations for C# programs from examples [196] to correct defects in student submissions, which however are mostly useless across assignments [147] and are not really defects in the wild as the violations in our study. Genesis [147] heuristically infers application-independent code transform patterns from multiple applications to fix bugs, but its code transform patterns are tightly coupled with the nature and syntax of three kinds of bugs (i.e., null pointer, out of bounds, and class cast defects). Koyuncu et al. [113] have generalized this approach with FixMiner to mining fix patterns for all types of bugs given a large dataset. Our work tries to mine the common fix patterns for general static analysis violations which are not application-independent. Closely related to our work is the concurrent work of Reudismam et al. [197] who try to learn quick fixes by mining code changes to fix PMD violations [69]. Their approach aims at learning code change templates to be systematically applied to refactor code. Our approach can be used for a similar scenario, and scales to a huge variety of violation types.

5.5.2 Program Repair

Recent studies of program repair have presented several achievements. There are mainly two lines of research: (1) fully automated repair and (2) patch hint suggestion. The former focuses on automatically generating patches that can be integrated into a program without human intervention. The patch generation process often includes patch verification to figure out whether the patch does not break the original functionality when it is applied to the program. The verification is often achieved by running a given test suite. Automatize violation repair is included in our future work. The latter techniques suggest code fragments that can help create a patch rather than generating

a patch ready to integrate. Developers may use the suggestions to write patches and verify them manually, that is similar to the patch generation of our work.

Fully Automated Repair Automated program repair is pioneered by GenProg [128, 224]. This approach leverages genetic programming to create a patch for a given buggy program. It is followed by an acceptability study [64] and systematic evaluation [126]. Regarding the acceptability issue, Kim et al. [104] advocated GenProg may generate nonsensical patches and proposed PAR to deal with the issue. PAR leverages human-written patches to define fix templates and can generate more acceptable patches. HDRepair [124] leverages bug fixing history of many projects to provide better patch candidates to the random search process. Recently, LSRepair [143] proposes a live search approach to the ingredients of automated repair using code search techniques. While GenProg relies on randomness, utilizing program synthesis techniques [164, 165, 179] can directly generate patches even though they are limited to a certain subset of bugs. Other notable approaches include contract-based fixing [223], program repair based on behavior models [51], and conditional statement repair [240]. This study does not focus on the fully automated program repair but the automated fix pattern mining for violations.

Patch Hint Suggestion Patch suggestion studies explored diverse dimensions. MintHint [100] generates repair hints based on statistical analysis. Tao et al. [216] investigated how automatically generated patches can be used as debugging aids. Bissyandé suggests patches for bug reports based on the history of patches [29]. Caramel [182] focuses on potential performance defects and suggests specific types of patches to fix those defects. Our study is closely related to patch hint suggestion since we can suggest top-10 most similar fix patterns for targeting violations. The difference is that fix patterns in this work are mined from developers' patch submissions of static analysis violations.

Empirical Studies on Program Repair Many studies have explored properties of program repair. Monperrus [173] criticized issues of patch generation learned from human-written patches [104]. Barr et al. discussed the plastic surgery hypothesis [22] that theoretically illustrates graftibility of bugs from a given program. Long and Rinard analyzed the search space issues for population-based patch generation [149]. Smith et al. presented an argument of overfitting issues of program repair techniques [202]. Koyuncu et al. [111] compared the impact of different patch generation techniques in Linux kernel development. Benchmarks for program repair are proposed for different programming languages [98, 127]. Based on a benchmark, a large-scale replication study was conducted [156]. More recently, Liu et al. [139] investigated the distribution of code entities impacted by bug fixes with fine-grained granularity, and found that some static analysis tools (e.g., FindBugs [90] and PMD [69]) are involved in some bug fixes.

5.6 Summary

In this study, we propose an approach to mining fix patterns of static analysis violations by leveraging CNNs and *X-means*. The identified fix patterns are evaluated through three experiments. They are first applied to fixing many unfixed violations in our subjects. Second, we manage to get 67 of 116 generated patches accepted by the developer community and eventually merged into 10 open source Java projects. Third, interestingly, the mined fix patterns were effective for addressing 4 real bugs in the Defects4J benchmark.

As further work, we plan to combine fix pattern mining with automated program repair techniques to generate violation fixes more automatically. In the live study, we find that some common violations never occurred in latest versions of those projects. We postulate that violation recurrences may be time-varying. Our future work also includes studies on the time-variant issue of violation recurrences to further figure out the historic changes of fixed violations and the latest trend of violations, which may help new directions of violation prioritization.

6 AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations

In this chapter, we propose to investigate the possibility in an APR scenario of leveraging code changes that address violations by static bug detection tools. To that end, we build the Avatar *APR system, which exploits fix patterns of static analysis violations as ingredients for patch generation. Evaluated on the Defects4J benchmark, we show that, assuming a perfect localization of faults,* Avatar *can generate correct patches to fix 34/39 bugs. We further find that* Avatar *yields performance metrics that are comparable to that of the closely-related approaches in the literature. While* Avatar *outperforms many of the state-of-the-art pattern-based APR systems, it is mostly complementary to current approaches. Overall, our study highlights the relevance of static bug finding tools as indirect contributors of fix ingredients for addressing code defects identified with functional test cases.*

This chapter is based on the work published in the following papers:

- Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, pages 1–12. IEEE, 2019

Contents

6.1 Overview

The current momentum of Automated Program Repair (APR) has led to the development of various approaches in the literature [43, 47, 102, 104, 122–124, 128, 147, 148, 150, 164, 179, 224, 239, 240]. In the software engineering community, the focus is mainly placed on fixing *semantic bugs*, i.e., those bugs that make the program behavior deviate from developer's intention [163, 179]. Such bugs are detected by test suites. APR researchers have then developed repair pipelines where program test cases are leveraged not only for localizing the bugs but also as the oracle for validating the generated patches.

Unfortunately, given that test suites can be incomplete, typical APR systems are prone to generate nonsensical patches that might violate the intended program behavior or simply introduce other defects which are not covered by the test suites [104]. A recent study by Smith et al. [202] has thoroughly investigated this issue and found that *overfitted* patches are actually common: these are patches that can pass all the available test cases, but are not actually *correct*.

To address the problem of patch correctness in APR, two research directions are being investigated in the literature. The first direction attempts to develop techniques for automatically augmenting the test suites [245]. The second one focuses on improving the patch generation process to reduce the probability of generating nonsensical patches [93, 113]. The scope of our work is the latter.

Mining fix templates from common patches is a promising approach to achieve patch correctness. As first introduced by Kim et al. [104], patch correctness can be improved by relying on fix templates learned from human-written patches. In their work, the template constructions were performed manually, which is a limiting factor and is further error-prone [173]. Since then, several approaches have been developed towards automating the inference of fix patterns from fix changes in developer code bases [93, 139, 147, 160, 219]. A key challenging step in the inference of patterns, however, is the identification and collection of a substantial set of relevant bug fix changes to construct the learning dataset. Patterns must further be precise and diverse to actually guarantee repair effectiveness.

There have been approaches to mining fix patterns and exploring the challenges in achieving the diversity and reliability of fix ingredients. Long et al. [147] have relied on only three simple bug types, while Koyuncu et al. [113] have focused on bug linking between bug tracking systems and source code management systems to identify probable bug fixes. Unfortunately, the former approach cannot find patterns to address a variety of bugs, while the latter may include patterns that are irrelevant to bug fixes since developer changes are not atomic [85]; it is thus challenging to extract useful and reliable patterns focusing on fix changes.

Our work proposes a new direction for pattern-based APR to overcome the limitations in finding reliable and diverse fix ingredients. Concretely, we focus on developer patches that are fixing static analysis violations. The advantages of this approach are: (1) the availability of toolsets for assessing whether a code change is actually a fix [21, 82], and (2) the ability to further pre-categorize the changes into groups targeting specific violations, leading to consistent fix patterns [137, 197]. Although static analysis violations (e.g., FindBugs [90] warnings) may appear irrelevant to the problem of *semantic* bug fixing, there are two findings in the literature, which can support our intuition of leveraging fix patterns from static analysis violation patches to address semantic bugs:

- *Locations of* **semantic bugs** *(unveiled through dynamic execution of test cases) can sometimes be detected by static analysis tools.* In a recent study, Habib et al. [76], have found that some bugs in the Defects4J dataset can be identified by static analysis tools: SpotBugs[1], FaceBook Infer[2] and Google ErrorProne[3]. Other studies [61, 172, 243] have also suggested that violations reported by static analysis tools might be smells of more severe defects in software programs.

[1] `https://spotbugs.github.io`
[2] `https://fbinfer.com`
[3] `https://errorprone.info`

- *Violation fix patterns have been used to successfully fix bugs in the wild.* In preliminary live studies, Liu et al. [137] and Rolim et al. [197] have shown that it can systematically fix statically detected bugs by using some of their previously-learned fix patterns. They further showed that project developers are even eager to integrate the systematization of such fixes based on the mined patterns.

> *Our work investigates to what extent fix patterns for static analysis violations can serve as ingredients to the patch generation step in an automated program repair pipeline.*

This paper thus makes the following contributions:

1. *We discuss a counterpoint to a recent study in the literature on the usefulness of static analysis tools to address real bugs.* We find that, although static bug finding tools, can detect a relatively small number of real-world semantic bugs from the Defects4J dataset, fix patterns inferred from the patches addressing statically detected bugs can provide relevant ingredients in an APR pipeline that is targeting semantic bugs.
2. *We propose* Avatar (static analysis violation fix pattern-based automated program repair), *a novel fix pattern-based approach to automated program repair.* Our approach differs from related work in the dataset of developer patches that is leveraged to extract fix ingredients. We build on patterns extracted from patches that have been verified (with bug detection tools) as true bug fix patches. Given the redundancy of bug types detected by static analysis tools, the associated fixes are intuitively more similar, leading to the inference of reliable common fix patterns. Avatar is available at: `https://github.com/SerVal-DTF/AVATAR`.
3. *We report on an empirical assessment of* Avatar *on the Defects4J benchmark.* We compare our approach with the state-of-the-art based on different evaluation aspects, including the number of fixed bugs, the exclusivity of fixed bugs, patch correctness, etc. Among several findings, we find that Avatar is capable of generating correct patches for 34 bugs, and partially-correct patches for 5 bugs, when assuming a perfect fault localization scenario.

6.2 Background

We provide background information on general pattern-based APR, as well as on pattern inference from static analysis violation data.

6.2.1 Automated Program Repair with Fix Patterns

Fix pattern-based APR has been widely explored in the literature [104, 124, 147, 150, 199, 213]. The basic idea is to abstract a code change into a *pattern* (interchangeably referred to as a *template*) that can be applied to a faulty code. The fixing process thus consists in leveraging context information of faulty code (e.g., abstract syntax tree (AST) nodes) to match context constraints defined in a given fix pattern. For example, the fix template "Method Replacer" provided in PAR [104] is presented as:

```
obj.method1(param) → obj.method2(param)
```

where the faulty method call `method1` is replaced by another method call `method2` with compatible parameters and return type. A method call is the context information for this template to match buggy code fragment. Thus, this template can be applied to any faulty statement that includes at least one method call expression. The template further guides the patch candidate generation where changes are proposed to replace the potentially faulty method call with another method call.

Mining fix patterns has some intrinsic issues. The first issue relates to the variety of patterns that must be identified to support the fixing of different bug types. There are two strategies in fix pattern mining: (1) manual design and (2) automatic mining. The former can effectively create precise fix patterns.

Unfortunately, it requires human effort, which can be prohibitive [104]. The latter infers common modification rules [147] or searches for the most redundant sub-patch instance [93, 113]. While this strategy can substantially increase the number of fix patterns, it is subject to noisy input data due to *tangled changes* [85], which make the inferred patterns less relevant. The second issue relates to the granularity (i.e., the degree of abstraction). Coarse-grained and monolithic patterns [185] can cover many types of bugs but they may not be actionable in APR. A fine-grained or micro pattern [147] can be readily actionable, but cannot cover many defects.

6.2.2 Static Analysis Violations

Static analysis tools help developers check for common programming errors in software systems. The targeted errors include syntactic defects, security vulnerabilities, performance issues, and bad programming practices. These tools are qualified as "static" because they do not require dynamic execution traces to find bugs. Instead, they are directly applied to source code or bytecode. In contrast with dynamic analysis tools which must run test cases, static tools can cover more paths, although it makes over-approximations that make them prone to false positives.

Many software projects rigorously integrate static analysis tools into their development cycles. The Linux kernel development project is such an example project where developers systematically run static analyzers against their code before pushing it to maintainers repositories [111]. More generally, FindBugs, PMD[4] and Google Error-Prone are often used in Java projects, while C/C++ projects tend to adopt Splint[5], cppcheck[6], and Clang Static Analyzer[7]. Static analysis tools raise warnings, which are also referred to as alerts, alarms, or violations. Given that these warnings are due to the detection of code fragments that do not comply with some analysis rules, in the remainder of this paper we refer to the issues reported by static analysis tools as *violations*.

Figure 6.1 shows an example patch for a violation detected by FindBugs. This violation (the red code) is reported because the `equals` method should work for all object types (i.e., `Object`): in this case, the method code violates the rule since it assumes a specific type (i.e., `ModuleWrapper`).

```
1  public boolean equals(Object obj) {
2      // Violation Type: BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS
3  -   return getModule().equals(((ModuleWrapper) obj).getModule());
4  +   return obj instanceof ModuleWrapper && getModule().equals(((ModuleWrapper)
                obj).getModule());
5  }
```

Figure 6.1: Example patch for fixing a violation detected by FindBugs.

Note that, not all violations are accepted by developers as actual defects. Since static analysis tools use limited information, detected violations could be correct code (i.e., false positive) or the warning may be irrelevant (e.g., cannot occur at runtime, or not a serious issue). In the literature, many studies assume that a violation can be classified as actionable if it is discarded after a developer changed the location where the violation is detected. The violation in Figure 6.1 is fixed by adding an `instanceof` check (c.f., the green code in the patch diff); this violation can thus be regarded as *actionable* since this violation is gone after fixing its source code.

Motivation Mining patterns from developer patches that fix static analysis violations may help overcome the issues of fix pattern mining described in Section 6.2.1. First, since static analysis tools specify the type of each violation (e.g., bug descriptions[8] of FindBugs), each bug instance is already

[4] https://pmd.github.io
[5] https://www.splint.org
[6] http://cppcheck.sourceforge.net
[7] https://clang-analyzer.llvm.org
[8] http://findbugs.sourceforge.net/bugDescriptions.html

classified as long as it is fixed by code changes. Thus, we can reduce the manual effort to collect and classify bugs and their corresponding patches for fix pattern mining. Second, we can mitigate the issue of tangled changes [85] because violation-fixing changes can be localized by static analysis tools [21]. Finally, the granularity of fix patterns can be appropriately adjusted for each violation type since static analysis tools often provide information on the scope of each violation instance.

6.3 Mining Fix Patterns for Static Violations

Mining fix patterns for static analysis violations has recently been explored in the literature [137, 197]. The general objective so far, however, is to learn quick fixes for speeding maintenance tasks and towards understanding which violations are prioritized by developers for fixing. To the best of our knowledge, our work is the first reported attempt to investigate fix patterns of static analysis violations in the context of automated program repair (where patches are generated and validated systematically with developer test cases).

There have been two recent studies of mining fix patterns addressing static analysis violations. Our previous study [137] focuses on identifying fix patterns for FindBugs violations [90], while Rolim et al. [197] consider PMD violations [48]. Both approaches, which were developed concurrently, leverage a similar methodology in the inference process. We summarise below the process of fix pattern mining of static analysis violations into three basic steps (as shown in Figure 6.2): data collection, data preprocessing, and fix pattern mining. Implementation details are strictly based on the approach of our previous study [137].

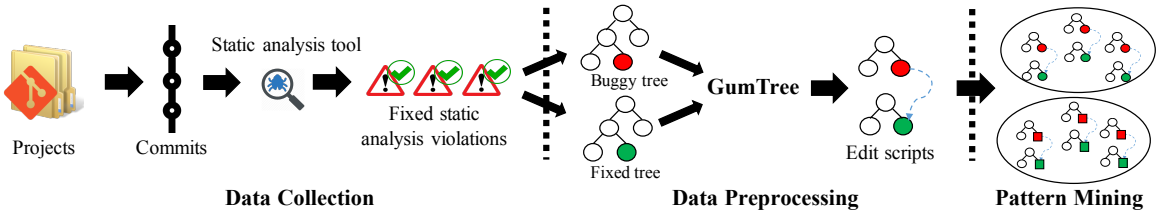


Figure 6.2: Summarized steps of static analysis violation fix pattern mining.

6.3.1 Data Collection

The objective of this step is to collect patches that are relevant to static analysis violations. This step is done in the wild based on the commit history of open-source projects by implementing a strict strategy to limit the dataset of changes to those that are relevant in the context of static analysis violations. To that end, it is necessary to systematically run static bug detection tools to each and every revision of the programs. This process can be resource-intensive: for example, FindBugs takes as input compiled versions of Java classes, requiring to build thousands of project revisions.

This step collects code changes (i.e., patches) only if they are identified as violation-fixing changes. For a given violation instance, we can assume that a change commit is a (candidate) fix for the instance when it disappears after the commit: i.e., the violation instance is identified in a revision of a program, but is no longer identified in the consecutive revision. Then, it is necessary to figure out whether the change actually fixed the violation instance or it just disappears by coincidence. If the affected code lines are located within the code change diff [9] of the commit, it is regarded as an actual fix for the given violation instance. Otherwise, the violation instance might be removed just by deleting a method, class, or even a file. Eventually, all code change diffs associated to the identified fixed violation instances are collected to form the input data for fix pattern mining.

[9] A "code change diff" consists of two code snapshots. One snapshot represents the code fragment that will be affected by a code change, while the other one represents the code fragment after it has been affected by the code change.

6.3.2 Data Preprocessing

Once violation patch data are collected, they are processed to extract concrete change actions. Patches submitted to program repositories are presented in the form of line-based GNU diffs where changes are reported in a text-based format of edit script. Given that, in modern programming languages, such as Java, source code lines do not represent a semantic entity of a code entity (e.g., a statement may span across several lines), it is challenging to directly mine fix patterns from GNU diffs.

Pattern-mining studies leverage edit scripts of program Abstract Syntax Trees (ASTs). Concretely, the buggy version (i.e., program revision file where the violation can be found) and the fixed version (i.e., consecutive program revision file where the violation does not appear) are given as inputs to the GumTree [60], an AST-based code differencing tool, to produce the relevant AST edit script. This edit script describes in a fine-grained manner the repair actions that are implemented in the patch. Figure 6.3 provides an example GNU Diff for a bug fix patch, while Figure 6.4 illustrates the associated AST edit scripts.

```
--- a/src/com/google/javascript/jscomp/Compiler.java
+++ b/src/com/google/javascript/jscomp/Compiler.java
@@ -1284,3 +1284,2 @@
   if (options.dependencyOptions.needsManagement() &&
-          !options.skipAllPasses && options.closurePass) {
+          options.closurePass) {

// Defects4J Dissection:
// Repair Action: Conditional expression reduction.
```

Figure 6.3: Patch of the bug Closure-13[10] in Defects4J.

```
UPD IfStatement@@"if statement code"
---UPD InfixExpression@@"infix-expression code"
------DEL PrefixExpression@@"!options.skipAllPasses"
------DEL Operator@@"&&"
```

Figure 6.4: AST edit scripts produced by GumTree for the patch in Fig. 6.3.

6.3.3 Fix Pattern Mining

Given a set of edit scripts, the objective of the pattern mining step is to group "similar" scripts in order to infer the common subset of edit actions, i.e., a consistent pattern across the group. To that end, Rolim et al. [197] rely on the greedy algorithm to compute the distance among edit scripts. Edit scripts with low distances among them are grouped together. Our previous study [137], on the other hand, leverage a deep representation learning framework (namely, CNNs [161]) to learn features of edit scripts, which are then used to find clusters of similar edit scripts. Clustering is performed based on the X-means algorithm. Finally, the largest common subset of edit actions among all edit scripts in a cluster is considered as the pattern.

Mined fix patterns with this approach have already been proven useful by the authors. For example, our previous study [137] and Rolim's work [197] conducted live studies by making pull requests to projects in the wild: the pull requests contained change details of a patch that is generated based on the inferred fix patterns to fix static analysis violations in developer code. Developers accepted to merge 67 out of 116 patches generated for FindBugs violations in our previous study [137]. Similarly, 6 out of 16 pull requests by Rolim et al. [197] have been merged by developers in the wild. Such promising results demonstrated the possibility to automatically fix bugs that are addressed by static bug detection tools.

[10] http://program-repair.org/defects4j-dissection/#!/bug/Closure/31

Fix patterns of static analysis violations have been explored in the literature to automate patch generation for bugs that are statically detected. To the best of our knowledge, AVATAR *is the first attempt to leverage fine-grained patterns of static analysis violations as fix ingredients for automated program repair that addresses semantic bugs revealed by test cases.*

6.4 Our Approach

As shown in Figure 6.5, AVATAR consists of four major steps for automated program repair. In this section, we detail the objective and design of each step, and provide concrete information on implementation.

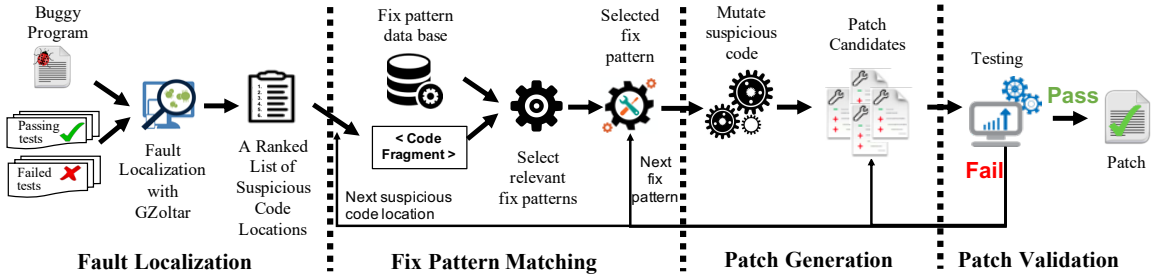


Figure 6.5: Overview bug fixing process with AVATAR.

6.4.1 Fault Localization

We rely on the GZoltar[11] [40] framework to automate the execution of test cases for each program. In the framework, we leverage the Ochiai [3] ranking metric to actually compute the suspiciousness scores of statements that are likely to be the faulty code locations. This ranking metric has been demonstrated in several empirical studies [187, 208, 236, 241] to be effective for localizing faults in object-oriented programs. The GZoltar framework for fault localization is also widely used in the literature of APR [93, 113, 143, 158, 226, 237, 239, 240].

6.4.2 Fix Pattern Matching

In the running of the repair pipeline, once fault localization produces a list of suspicious code locations, AVATAR iteratively attempts to match each of these locations with a given pattern from the database of mined fix patterns. Fix patterns in our database are collected from the artifacts released by Liu et al. [137] and Rolim et al. [197]. Table 6.1 shows statistics about the pattern collection in these previous works. As most of the fix patterns released by Liu et al. will not change the program behavior, we only select 13 of them for 10 violation types after manually checking that they can change the program behavior (details shown in the aforementioned website).

Table 6.1: Statistics on fix patterns of static analysis violations.

	# Projects	# violation fix patches	# violation types	# fix patterns
Liu et al. [137]	730	88,927	111	174
Rolim et al. [197]	9	288,899	9	9

Recall that each pattern is actually an edit script of repair actions on specific AST node types. AST nodes associated to the faulty code locations are then regarded as the *context* of matching the fixing patterns: i.e., these nodes are checked against the nodes involved in the edit scripts of fix patterns.

[11] http://www.gzoltar.com

For example, the fix pattern shown in Figure 6.6 contains three levels of contexts: (1) `IfStatement`, which means that the pattern is matched only if the suspicious faulty statement is an `IfStatement`; (2) `InfixExpression` indicates that the pattern is relevant when the predicate expression of the suspicious `IfStatement` is an `InfixExpression`; (3) the matched `InfixExpression` predicate in the suspicious statement must contain at least two sub-predicate expressions.

```
1 // Fix Pattern:  Remove a useless sub-predicate expression.
2 UPD IfStatement
3 ---UPD InfixExpression@expA Operator expB
4 ------DEL Expression@expB
5 ------DEL Operator
```

Figure 6.6: A fix pattern for `UC_USELESS_CONDITION`[12] violation [137].

A pattern is found to be relevant to a faulty code location only if all AST node contexts at this location matches with the AST node of the pattern. For example, the bug shown in Figure 6.3 is located within an `IfStatement` with an `InfixExpression` which is formed by three sub-predicate expressions. This buggy fragment thus matches the fix pattern shown in Figure 6.6.

6.4.3 Patch Generation

Given a suspicious code location and an associated matching fix pattern, AVATAR applies the repair actions in the edit scripts of the pattern to generate patch candidates. For example, the code change action of the fix pattern in Figure 6.6 is interpreted as removing a sub-condition expression (or sub-predicate expression) in a faulty `IfStatement`. Thus, three patch candidates, as shown in Figure 6.7, can be generated by AVATAR for the buggy code in Figure 6.3 since the statement has three candidate sub-predicates expressions.

```
1   // Patch Candidate I.
2 - if (options.dependencyOptions.needsManagement() &&}
3 -           !options.skipAllPasses && options.closurePass) {
4 + if (!options.skipAllPasses && options.closurePass) {
5
6   // Patch Candidate II.
7   if (options.dependencyOptions.needsManagement() &&
8 -           !options.skipAllPasses && options.closurePass) {
9 +           options.closurePass) {
10
11  // Patch Candidate III.
12  if (options.dependencyOptions.needsManagement() &&
13 -           !options.skipAllPasses && options.closurePass) {
14 +           !options.skipAllPasses) {
```

Figure 6.7: Patch Candidates generated by AVATAR with a fix pattern that is mined from patches for `UC_USELESS_CONDITION` violations (cf. Figure 6.6), and which matches the buggy statement in Closure-13 bug (cf. Figure 6.3).

6.4.4 Patch Validation

Patch candidates generated by AVATAR must then be systematically assessed. Eventually, using test cases, our approach verifies whether a patch candidate is a *plausible* patch or not. We target two types of plausible patches:

[12]The condition has no effect and always produces the same result as the value of the involved variable was narrowed before. Probably something else was meant or condition can be removed.

- ***Fully-fixing*** patches, which are patches that make the program pass all available test cases. Once such a patch is validated, the execution iterations of AVATAR are halted.
- ***Partially-fixing*** patches, which are patches that make the program pass not only all previously-passing test cases, but also part of the previously-failing test cases.

The first generated *fully-fixing* patch is prioritized over any other generated patch, and is considered as the plausible patch for the given bug. After iterating over all suspicious statements with all matching fix patterns, if AVATAR fails to generate a *fully-fixing* patch for a bug, but generates some *partially-fixing* patches, these patches are considered as plausible patches. Nevertheless, we implement a selection scheme where *partially-fixing* patches that change the program control-flow are prioritized over those that only change data-flow.

Partially-fixing patches that change the control flow are further naïvely ordered by the edit distances (at AST node level) between the patched fragments and the buggy fragment: smaller edit distances are preferred. Ties are broken by considering precedence in the generation: the first generated *partially-fixing* patch is the final plausible patch.

6.5 Assessment

We evaluate AVATAR on Defects4J [98], which is widely used by state-of-the-art APR systems targeting Java programs. Table 6.2 summarizes the statistics on the number of bugs and test cases available in the version 1.2.0[13] of Defects4J.

Table 6.2: Defects4J dataset information.

Project	Bugs	kLoC	Tests
JFreeChart (Chart, C)	26	96	2,205
Closure compiler (Closure, Cl)	133	90	7,927
Apache commons-lang (Lang, L)	65	22	2,245
Apache commons-math (Math, M)	106	85	3,602
Mockito (Moc)	38	11	1,457
Joda-Time (Time, T)	27	28	4,130
Total	395	332	21,566

"**Bugs**", "**kLoC**", and "**Tests**" denote respectively the number of bugs, the program size in kLoC (i.e., thousands of lines of code), and the number of test cases for each subject. The overall number of kLoC and test cases for project Mockito are not indicated in the Defects4J paper [98] from where the reported information is excerpted.

6.5.1 Research Questions

Our investigation into the repair performance of AVATAR seeks to answer the following research questions (RQs):

- **RQ1:** ***How effective are fix patterns of static analysis violations for repairing programs with semantic bugs?*** Recall that we broadly consider as *semantic bugs* all bugs that are uncovered by executing developer *test cases*. Our first research question assesses how many bugs in the Defects4J benchmark can be fixed with fix patterns of static analysis violations. To that end, we first (1) investigate how effectively AVATAR can fix such semantic bugs that appear to be localizable by static analysis tools. Then, (2) building on the assumption that the location of the semantic bug is known, we investigate whether AVATAR can generate a correct patch to fix it.

[13] `https://github.com/rjust/defects4j/releases/tag/v1.2.0`

- **RQ2:** ***Which bugs and which patterns are relevant targets for* Avatar *in an automated program repair scenario?*** This research question dissects the data yielded during the investigation of RQ1, with the objective of assessing the diversity of bugs that can be fixed as well as the types of violation fix patterns that have been successfully leveraged.
- **RQ3:** ***How does* Avatar *compare to the state-of-the-art with respect to repair performance?*** With this research question, we aim at showing whether the proposed approach is relevant in the landscape of APR systems. Does AVATAR offer comparable performance? To what extent can AVATAR complement existing APR systems?

6.5.2 Experimental Setup

For evaluation purpose, we apply different fault localization schemes to the experiment of each RQ, while the default setting of AVATAR is to use the GZoltar framework with the Ochiai ranking metric for ordering suspicious statements. The usage of GZoltar and Ochiai reduces the comparison biases since both are widely used by APR systems in the literature.

- First, we apply AVATAR to Defects4J bugs that are localized by three state-of-the-art static analysis tools (namely, SpotBugs, Facebook Infer, and Google ErrorProne) (for RQ1; see Section 6.5.3). To that end, we consider recent data reported by Habib and Pradel [76]. This configuration focuses on the effectiveness of AVATAR on such semantic bugs that can also be detected statically.
- Second, we apply AVATAR on all faulty code positions in the benchmark (for RQ1; see Section 6.5.4). We thus assume that a perfect localization is possible, and assess the performance of the approach on all bugs.
- Finally, for RQ3, we compare AVATAR with the state-of-the-art APR tools that are evaluated on the Defects4J benchmark (see Section 6.5.6). To that end, we attempt to replicate two scenarios of fault localization used in APR assessments: the first scenario assumes that the faulty method name is known [124] and thus focuses on ranking the inner-statements based on Ochiai suspiciousness scores; the second scenario makes no assumption on fault location and thus uses the default setting of AVATAR.

6.5.3 Applying Avatar to Statically-Detected Bugs in Defects4J

Table 6.3 provides details on the Defects4J bugs that can be detected by static analysis tools and are successfully repaired by AVATAR. We report that out of the 14, 4, and 7 bugs that can be detected respectively by SpotBugs, Facebook Infer and Google ErrorProne on version 1.2.0 of Defects4J, AVATAR can successfully generate correct patches for 3, 2 and 1 bugs.

Overall four distinct localizable bugs have been correctly fixed with patches generated from the fix patterns that were mined from patches fixing FindBugs violations [137]. All the four bugs are related to distinct violation types. In their work, Liu et al. [137] claimed that their mined patterns could be applied to violations reported by other static analysis tools. Our experiment indeed shows that these patterns fixed two bugs detected by SpotBugs (i.e., the successor of FindBugs), which are also detected by Facebook Infer.

> ***RQ1*** ▶ AVATAR *demonstrates the usefulness of violation fix patterns in a scenario of automating some maintenance tasks involving systematic checking of developer code with static checkers.*

[14] A null pointer is dereferenced and will lead to a NullPointerException when the code is executed.
[15] Possible null pointer dereference.
[16] Dead store to a local variable.
[17] A private method is never called.
[18] `https://github.com/rjust/defects4j/pull/112`

Table 6.3: Statically-detected bugs fixed by AVATAR.

Bug ID	SpotBugs	Infer	ErrorProne	Static Analysis Violation Type
Chart-1	●	●		NP_ALWAYS_NULL[14]
Chart-4	●	●		NP_NULL_ON_SOME_PATH[15]
Chart-24	●			DLS_DEAD_LOCAL_STORE[16]
Math-77			●	UPM_UNCALLED_PRIVATE_METHOD[17]
Total	3/14 (18)	2/4 (5)	1/7 (8)	

$x/y(z)$ reads as: x is the number of bugs fixed by AVATAR among the y bugs in version 1.2.0 of Defects4J that can be localized by each static analysis tool. z is given as an indicator for the number of statically localizable bugs in an augmented version[18]of Defects4J. The information on localizable bugs is excerpted from the study of Habib and Pradel [76]. Since most of the state-of-the-art APR systems targeting Java program are evaluated on the version 1.2.0, our experiments focus on localizable bugs in this version.

Our experiments, however, reveal that AVATAR's fix patterns for static analysis violations are not effective on many supposedly statically-detectable bugs in Defects4J. We investigate the cases of such bugs, and find that:

1. some of the bugs have a code context that does not match any of the mined fix patterns;
2. in some cases, the detection is actually a coincidental false positive reported in [76], since the violation does not really match the semantically faulty code entity that must be modified. Figure 6.8 provides a descriptive example of such false positives.
3. finally, in other cases, the fixes are truly domain-specific, and no pattern is thus applicable.

```
1 // Violation Type:  FE_FLOATING_POINT_EQUALITY
2 // The comparing result of two floating point values for
3 // equality may not be accurate.
4
5 // Defects4J Dissection:
6 // Bug Pattern: Conditional block removal.
7
8 - if (x == x1) {
9 -         x0 = 0.5 * (x0 + x1 - FastMath.max(rtol * FastMath.abs(x1), atol));
10 -        f0 = computeObjectiveValue(x0);
11 - }
12   break;
```

Figure 6.8: A bug is false-positively identified as a statically-detected bug in [76] (Math-50): the violation is not related to the test case failures.

6.5.4 Applying Avatar to All Defects4J Bugs

After focusing on those Defects4J bugs that can be statically detected, we run AVATAR on all the dataset bugs. Given that the objective is to assess whether a correct patch can be generated if the bug is known, we assume in this experiment that the faulty code locations are known. Concretely, we do not rely on any fault localization tool. Instead, we consider the ground truth of developer patches and list the locations that have been modified as the faulty locations.

The repair operations thus consist in generating patches for the relevant bug locations. Table 6.4 details the number of Defects4J bugs that are fixed by AVATAR. Fully and partially fixed bugs are fixed with *fully-fixing* and *partially-fixing* patches (c.f. Section 6.4.4) generated by AVATAR, respectively. Overall, AVATAR can fix 49 bugs with plausible patches (i.e., that pass all available tests). 35 of them are further manually confirmed to be *correct* (i.e., they are syntactically or at least semantically

equivalent to the ground truth patches submitted by developers). We also note that, for 14 other bugs, AVATAR generates plausible patches that make the program pass some previously-failing test cases, without failing any of the previously-passing test cases. Five among these patches are manually found to be correctly fixing part of the bugs. To the best of our knowledge, AVATAR is the first APR tool which partially, but correctly, fixes bugs in Defects4J that have multiple fault code locations.

Table 6.4: Number of Defects4J bugs fixed by AVATAR with an assumption of perfect localization information.

Fixed Bugs	Chart	Closure	Lang	Math	Mockito	Time	Total
# of Fully Fixed Bugs	7/8	10/13	5/10	8/13	2/2	2/3	34/49
# Partially Fixed Bugs	2/3	1/4	1/3	1/4	0/0	0/0	5/14

†In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool. The same as the following similar tables.

RQ1▶AVATAR *can effectively fix several semantic bugs from the Defects4J dataset. We even observe that the fine-grained fix ingredients can be helpful to target bugs with multiple faulty code locations.*

6.5.5 Dissecting the Fix Ingredients

We now investigate how fix patterns of static analysis violations can be leveraged to address semantic bugs from the Defects4J benchmark. To that end, we dissect the ingredients that were successfully leveraged in the generated correct patches. Table 6.5 provides the summary of this dissection.

Table 6.5: Fix ingredients leveraged in the static analysis violation fix patterns used by AVATAR to correctly fix semantic bugs.

Violation Types associated with the Fix Patterns	Fix Ingredients from the Violation Fix Patterns	Fixed Bug IDs
NP_ALWAYS_NULL	Mutate the operator of null-check expression: “ var != == null”, or “var == != null”.	C-1.
NP_NULL_ON_SOME_PATH	1. Wrap buggy code with if-non-null-check block: “if (var != null) {buggy code}”; 2. Insert if-null-check block before buggy code: “if (var == null) {return false;} buggy code;” or “if (var == null) {return null;} buggy code;” or “if (var == null) {throw IllegalArgumentException.} buggy code;”.	C-4,26, C-14,19, *C-25*,Cl-2, M-4, Moc-29,38.
DLS_DEAD_LOCAL_STORE	Replace a variable with other one: e.g., “var1 = var2var3;”.	C-11,24, L-6,57,59, M-33,59,T-7.
UC_USELESS_CONDITION	1. Mutate the operator of an expression in an if statement: e.g., “if (expA >>= expB) {...}”; 2. Remove a sub-predicate expression in an if statement: “if (expA || expB) {...}” or “if (expA || expB) {...}”; 3. Remove the conditional expression: “expA ? expB : expC” or “expA ? expB : expC”.	Cl-18,31, Cl-38,62, Cl-63,73, *L-15*,M-46, M-82,85, T-19.
UCF_USELESS_CONTROL_FLOW[19]	1. Remove an if statement but keep the code inside its block: “if (exp) { code }”; 2. Remove an if statement with its block code: “if (exp) { code }”.	*C-18*, *Cl-106*, Cl-115,126, L-7,10, M-50.
UPM_UNCALLED_PRIVATE_METHOD	Remove a method declaration: “Modifier ReutrnType methodName(Parameters) { code }”.	Cl-46,*M-77*.
BC_UNCONFIRMED_CAST[20]	Wrap buggy code with if-instanceof-check block: “if (var instanceof Type) {buggy code} else {throw IllegalArgumentException;}”.	M-89.

† Only correctly fixed (including partially correctly-fixed bugs highlighted with *italic*) bugs are listed in this table.

First, we note that all correctly fixed bugs were addressed with patches generated from patterns mined in the study of Liu et al. [137] (i.e., based on FindBugs violations). Fix patterns from the study by Rolim et al. [197] (which are based on PMD violations) are indeed associated to exceedingly

Table 6.6: Dissection of bugs correctly fixed by AVATAR.

Bug IDs	Bug Patterns	Repair Actions
C-1.	Conditional expression modification	Logic expression modification.
C-4, 26.	Missing non-null check addition	Conditional (if) branch addition.
C-14, 19, 25, Cl-2, M-4, Moc-29, 38.	Missing null check addition	Conditional (if) branch addition.
C-11, 24, L-6, 57, 59, M-33, 59, T-7.	Wrong variable reference	Variable replacement by another variable.
Cl-38.	Logic expression expansion	Conditional expression expansion.
Cl-18, 31, L-15.	Logic expression reduction	Conditional expression reduction.
Cl-62, 63, 73, M-82, 85, T-19.	Logic expression modification	Conditional expression modification.
C-18, Cl-106, M-46.	Unwraps-from if-else statement	Conditional (if or else) branch removal.
Cl-115, 126, L-7, 10, M-50.	Conditional block removal	Conditional (if or else) branch removal.
Cl-46, M-77.	Unclassified	Method definition removal.
M-89.	Wraps-with if-else statement	Conditional (if-else) branches addition.

simple violations, which are unlikely to be revealed as semantic bugs (i.e., detected via developer test cases). An example of such simple pattern is their EP7 fix pattern: "replace `List<String> a = new ArrayList()` with `List<String> a = new ArrayList<>()`". In any case, 6 among the 9 fix patterns released by Rolim et al. are related to performance, code practice or code style. Our manual investigation of Defects4J bugs reveals that none of the bugs are associated to these types of issues.

Second, we note that the fix patterns of only seven (out of 10) violation types have been successfully used to generate correct patches for Defects4J bugs (c.f. Table 6.5). Among the 40 (fully or partially) correctly fixed bugs, 36 (90%) are fixed with fix patterns of 4 violation types: `NP_ NULL_ON_SOME_PATH`, `DLS_DEAD_LOCAL_STORE`, `UC_ USELESS_CONDITION`, and `UCF_USELESS_CONTROL_ FLOW`. The latter two violation types are related to the issues of conditional code entities (e.g., If statements and conditional expressions), which are relevant to 18 (45%) of the fixed bugs. Comparing against the ACS [239] state-of-the-art APR tool which focuses on repairing condition-related faulty code entities, we find that AVATAR correctly fixes 15 relevant bugs that are not fixed by ACS.

Finally, we investigate the diversity of the bugs that are correctly addressed by AVATAR. To that end, we resort to the dissection study of Defects4J bugs by Sobreira et al. [203]. Table 6.6 summarizes the bug patterns and the associated repair actions for the bugs that are correctly fixed by AVATAR. We note that AVATAR can currently address 11 bug patterns out of the 60 bug patterns enumerated in the dissection study.

> ***RQ2***►AVATAR *exploits a variety of fix ingredients from static violations fix patterns to address a diverse set of bug patterns in the Defects4J dataset.*

6.5.6 Comparing Against the State-of-the-Art

To reliably compare against the state-of-the-art in Automated Program Repair (APR), we must ensure that the Fault Localization (FL) step is properly tuned as FL could bias the bug fixing performance of APR tools [140]. We identify three major configurations in the literature:

1. ***Normal_FL*-based APR**: in this case, APR systems directly use off-the-shelf fault localization techniques to localize the faulty code positions. In this case, which is realistic, the suspicious list of fault locations may be inaccurate leading to a poor repair performance. APR tools that are run in ASTOR [158] work under this configuration.
2. ***Restricted_FL*-based APR**: in this case, APR systems make the assumption that some information of the code location is available. For example, in HDRepair [124], fault localization is restricted to the faulty methods, which are assumed to be known. Such a restriction actually substantially increases the accuracy of the target list of fault locations for which a patch must be generated. In Section 6.5.4, we have made a similar strong assumption that fault locations are known as our objective was to assess the patch generation performance of Avatar.
3. ***Supplemented_FL*-based APR**: in this case, APR systems leverage existing fault localization tools but improve the localization approach with some heuristics to ensure that the patch generation targets an accurate list of code locations. For example, the SimFix [93] recent state-of-the-art system employs a test purification [242] technique to improve the accuracy of the fault localization.

We thus compare the bug fixing performance of Avatar with the state-of-the-art APR tools after classifying them into these three groups.

6.5.6.1 Comparison against a Restricted_FL-based APR System

We first compare Avatar against the HDRepair [124] state-of-the-art APR system, which implements a *restricted* fault localization configuration. We select faulty locations using the same assumption as HDRepair, i.e., focusing on attempting to repair suspicious code statements that are reported by our fault localization tool but filtering only those that are within the known faulty methods. This assumption leaves out many noisy statements, reducing the probability of generating overfitting patches for bugs and further increasing the chance to generate a correct patch before a plausible one or any execution timeout.

Table 6.7: Comparison of Avatar with HDRepair [124].

Project	HDRepair	Avatar	
		Fully fixed	Partially fixed
Chart	0/2	7/9	1/3
Closure	0/7	9/15	1/2
Lang	2/6	5/12	1/3
Math	4/7	6/14	1/3
Mockito	0/0	2/2	0/0
Time	0/1	2/3	0/0
Total	6/23	31/55*	4/11*
P (%)	26.1	56.4	36.4

The results for HDRepair are provided by its author.
*The number of bugs fixed by Avatar shown in this table is a little different from the data in Table 6.4. For fixing each bug, the input of Avatar is a ranked list of suspicious statements in the faulty methods, which is different from the input of Avatar in the experiment of Section 6.5.4.

Table 6.7 presents the comparing results. Comparing with HDRepair, Avatar correctly fixes many more bugs (31+4 vs 6) and yields a higher probability to generate correct patches among all plausible patches (cf. P(%) in Table 6.7). 34 (except *Lang-6*) out of the 35 bugs fixed by Avatar are not addressed by HDRepair. Avatar also correctly fixes 7 bugs (as highlighted with **bold** in Figure 6.9)

[19]The current code contains a useless control flow statement, where the control flow continues onto the same place regardless of whether or not the branch is taken.
[20]The cast expression is unchecked or unconfirmed, and not all instances of the type casted from can be cast to the type it is being cast to. It needs to check that the program logic ensures that this cast will not fail.

that are only plausibly (but incorrectly) fixed by HDRepair. Finally, AVATAR partially fixes 11 bugs that have multiple faulty code fragments, and 4 of the associated patches are correct. Figure 6.9 illustrates the space of correctly fixed bugs by HDRepair and AVATAR.

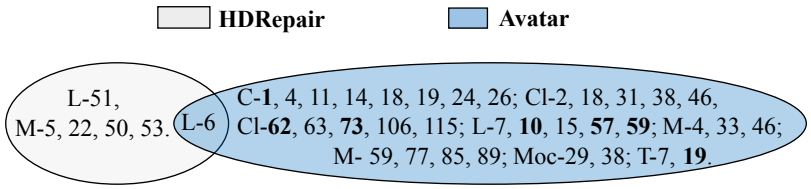


Figure 6.9: Bugs correctly fixed by HDRepair and AVATAR, respectively.

RQ3 ▶ AVATAR *substantially outperforms the HDRepair approach on the Defects4J benchmark.*

6.5.6.2 Comparison Against Normal_FL-based APR Systems

We compare the bug fixing performance of AVATAR with the *Normal_FL*-based state-of-the-art APR tools that are evaluated on the Defects4J benchmark. These APR tools take as input a ranked list of suspicious statements that are reported by an off-the-shelf fault localization technique. In this experiment, we consider a group of APR systems, namely jGenProg [156], jKali [156], jMutRepair [158], Nopol [240], FixMiner [113] and LSRepair [143], which leverage a similar configuration as AVATAR for fault localization: GZoltar/Ochiai.

Table 6.8 reports the comparison results in terms of the number of *plausibly-fixed* bugs and the number of *correctly-fixed* bugs. Data about the fixed bugs are directly excerpted from the results reported in the relevant research papers. We note that AVATAR outperforms all of the *Normal_FL*-based APR systems, both in terms of the number of plausibly fixed bugs and the number of correctly fixed bugs. It also yields a higher probability to generate correct patches among its plausible patches than those tools (except FixMiner). Finally, 18 (15 + 3, as shown in the second row of Table 6.9) among the 30 (27 + 3) bugs correctly fixed by AVATAR have not been correctly fixed by those *Normal_FL*-based APR tools.

Table 6.8: Number of bugs reported as having been fixed by different APR systems.

Fault Localization	APR Tool		Chart	Closure	Lang	Math	Mockito	Time	Total	P*(%)
Normal_FL-based APR	**Avatar**	Fully Fixed	**5/12**	**8/12**	5/11	6/13	**2/2**	1/3	27/53*	50.9
		Partially Fixed	1/2	1/1	0/2	1/3	0/0	0/0	3/8*	37.5
	jGenProg [158]		0/7	0/0	0/0	5/18	0/0	0/2	5/27	18.5
	jKali [158]		0/6	0/0	0/0	1/14	0/0	0/2	1/22	4.5
	jMutRepair [158]		1/4	0/0	0/1	2/11	0/0	0/1	3/17	17.6
	Nopol [240]		1/6	0/0	3/7	1/21	0/0	0/1	5/35	14.3
	FixMiner [113]		5/8	5/5	2/3	12/14	0/0	1/1	25/31	80.65
	LSRepair [143]		3/8	0/0	8/**14**	7/14	1/1	0/0	19/37	51.4
Supplemented_FL-based APR	**ACS** [239]		2/2	0/0	3/4	12/16	0/0	1/1	18/23	78.3
	ELIXIR [199]		4/7	0/0	8/12	12/19	0/0	**2**/3	26/41	63.4
	JAID [43]		2/4	5/11	1/8	1/8	0/0	0/0	9/31	29.0
	ssFix [237]		3/7	2/11	5/12	10/**26**	0/0	0/**4**	20/**60**	33.3
	CapGen [226]		4/4	0/0	5/5	12/16	0/0	0/0	21/25	**84.0**
	SketchFix [91]		**6**/8	3/5	3/4	7/8	0/0	0/1	19/26	73.1
	SimFix [93]		4/8	6/8	**9**/13	**14/26**	0/0	1/1	**34**/56	60.7

"**P**" is the probability of generated plausible patches to be correct.
*The number of bugs fixed by AVATAR shown in this table is a little different from the data in Table 6.4 and Table 6.7. In this experiment, for fixing each bug, the input of AVATAR is a ranked full list of suspicious statements in the faulty program, which is different from the input of AVATAR in the experiments of Table 6.4 and Table 6.7.

> ***RQ3***▶*In terms of quantity and quality of generated plausible patches,* AVATAR *addresses more bugs than its immediate competitors. Nevertheless, we note that* AVATAR *is actually complementary to the other state-of-the-art APR systems, fixing bugs that others do not fix.*

6.5.6.3 Comparison against Supplemented_FL-based APR Systems

We also compare AVATAR against APR systems which use supplementary information to improve fault localization accuracy. We include in this category other APR systems whose authors do not explicitly describe the actual fault localization configuration, but which still manage to fix bugs that we could not localize with GZoltar/Ochiai. We include in this group the following state-of-the-art works targeted at Java programs: ACS [239], ELIXIR [199], JAID [43], ssFix [237], CapGen [226], SketchFix [91] and SimFix [93].

Table 6.9: Bugs fixed by AVATAR but not correctly fixed by other APR tools.

APR tool group	Bug IDs	
	Fully-fixed	Partially-fixed
Normal_FL-based APR tools	C-14,19,Cl-2,18,31,46,L-6,7,10,M-4,46,59,Moc-29,38,T-7.	C-18, Cl-106, M-77.
Supplemented_FL-based APR tools	C-4,Cl-2,31,38,46,L-6,7,10,M-46,Moc-29,38.	C-18, Cl-106, M-77.
All APR tools	Cl-2,31,46,L-7,10,M-46,Moc-29,38.	C-18, Cl-106, M-77.

The compared performance results are also illustrated in Table 6.8. Based on the number of correctly fixed bugs, AVATAR is only inferior to SimFix but outperforms other *Supplemented_FL*-based APR systems. AVATAR further correctly fixes 14 (11 + 3, as shown in the third row of Table 6.9) out of 31 bugs that have never been addressed by any *Supplemented_FL*-based state-of-the-art APR system.

To sum up, AVATAR correctly fixes 11 (as shown in the fourth row of Table 6.9) out of 31 bugs that have never been addressed by any state-of-the-art APR system. We also note that AVATAR outperforms state-of-the-art APR tools on fixing bugs in project *Chart*, *Closure* and *Mockito*.

> ***RQ3***▶AVATAR *underperforms against some of the most recent APR systems. Nevertheless,* AVATAR *is still complementary to them as it is capable of addressing some Defects4J bugs that the state-of-the-art cannot fix.*

6.6 Threats to Validity

A threat to external validity is related to use of Defects4J bugs as a representative set of semantic bugs. This threat is mitigated as it is currently a widely used dataset in the APR literature related to Java. A threat to internal validity is due to the use of Java programs as subjects. Eventually, we only considered fix patterns for FindBugs and PMD violations. Other static tools, especially for C programs, such as Splint, cppcheck, and Clang Static Analyzer are not investigated. A threat to construct validity may involve the assumption of perfect localization to assess AVATAR. This threat is minimized by the different other experiments that are comparable with evaluations in the literature.

6.7 Related Work

The software development practice is increasingly accepting generated patches [111]. Recently, various directions in the literature have been explored to contribute to the advancement of automated program

repair. One commonly studied direction is the pattern based (also called example-based) APR. Kim et al. [104] initiated with PAR a milestone of APR based on fix templates that were manually extracted from 60,000 human-written patches. Later studies [124] have shown that the six templates used by PAR could fix only a few bugs in Defects4J. Long and Rinard also proposed a patch generation system, Prophet [150], that learns code correctness models from a set of successful human patches. They further proposed a new system, Genesis [147], which can automatically infer patch generation transforms from developer submitted patches for program repair.

Motivated by PAR [104], more effective automated program repair systems have been explored. HDRepair [124] was proposed to repair bugs by mining closed frequent bug fix patterns from graph-based representations of real bug fixes. Nevertheless, its fix patterns, except the fix templates from PAR, still limits the code change actions at abstract syntax tree (AST) node level, but are not specific for some types of bugs. ELIXIR [199] aggressively uses method call related templates from PAR with local variables, fields, or constants, to construct more expressive repair-expressions that go into synthesizing patches.

Tan et al. [213] integrated anti-patterns into two existing search-based automated program repair tools (namely, GenProg [128] and SPR [148]) to help alleviate the problem of incorrect or incomplete fixes resulting from program repair. In their study, the anti-patterns are defined by themselves and limited to the control flow graph. Additionally, their anti-patterns are not meant to solve the problem of deriving better patches automatically, provide more precise repair hints to developers.

More recently, CapGen [226], SimFix [93], FixMiner [113] are further proposed to fix bugs automatically based on the frequently occurred code change operations (e.g., Insert IfStatement (c.f., Table 3 in [93]) that are extracted from the patches in developer change histories.

So far however, pattern-based APR approaches focus on leveraging patches that developer applied to semantic bugs. To the best of our knowledge, our approach is first to investigate the case of leveraging patches that fix static analysis violations: they are many more, better identifiable, and more consistent.

6.8 Summary

The correctness of patches generated is now identified as a barrier in the adoption of automated program repair systems. Towards guaranteeing correctness, researchers have been investigating example-based approaches where fix patterns from human patches are leveraged in patch generation. Nevertheless, such ingredients are often hard to collect reliably. In this work, we propose to rely on developer patches that address static analysis bugs. Such patches are concise and precise, and their efficacy (in removing the bugs) are systematically assessed (by the static detectors). We build Avatar, an APR system that utilizes fix ingredients from static analysis violations patches. We empirically show that Avatar is indeed effective in repairing programs that have semantic bugs. Avatar outperforms several state-of-the-art approaches and complements others by fixing some of the Defects4J bugs which were not yet fixed by any APR system in the literature.

7 TBar: Revisiting Template based Automated Program Repair

In this stduy, we revisit the performance of template-based APR to build comprehensive knowledge about the effectiveness of fix patterns, and to highlight the importance of complementary steps such as fault localization or donor code retrieval. To that end, we first investigate the literature to collect, summarize and label recurrently-used fix patterns. Based on the investigation, we build `TBar`*, a straightforward APR tool that systematically attempts to apply these fix patterns to program bugs. We thoroughly evaluate* `TBar` *on the Defects4J benchmark. In particular, we assess the actual qualitative and quantitative diversity of fix patterns, as well as their effectiveness in yielding plausible or correct patches. Eventually, we find that, assuming a perfect fault localization,* `TBar` *correctly/plausibly fixes 74/101 bugs. Replicating a standard and practical pipeline of APR assessment, we demonstrate that* `TBar` *correctly fixes 43 bugs from Defects4J, an unprecedented performance in the literature (including all approaches, i.e., template-based, stochastic mutation-based or synthesis-based APR).*

This chapter is based on the work published in the following research paper:

- Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42. ACM, 2019

Contents

7.1 Overview

Automated Program Repair (APR) has progressively become an essential research field. APR research is indeed promising to improve modern software development by reducing the time and costs associated with program debugging tasks. In particular, given that faults in software cause substantial financial losses to the software industry [31, 181], there is a momentum in minimizing the time-to-fix intervals by APR. Recently, various APR approaches [43, 47, 91, 93, 102, 104, 122–124, 128, 138, 140, 141, 147, 148, 150, 164, 179, 224, 226, 239, 240] have been proposed, aiming at reducing manual debugging efforts through automatically generating patches.

An early strategy of APR is to generate concrete patches based on fix patterns [104] (also referred to as fix templates [144] or program transformation schemas [91]). This strategy is now common in the literature and has been implemented in several APR systems [56, 91, 104, 113, 140, 141, 144, 159, 199]. Kim et al. [104] showed the usefulness of fix patterns with *PAR*. Saha et al. [199] later proposed *ELIXIR* by adding three new patterns on top of *PAR* [104]. Durieux et al. [56] proposed *NPEfix* to repair null pointer exception bugs, using nine pre-defined fix patterns. Long et al. designed *Genesis* [147] to infer fix patterns for specific three classes of defects. Liu and Zhong [144] explored posts from Stack Overflow to mine fix patterns for APR. Hua et al. proposed *SketchFix* [91], a runtime on-demand APR tool with six pre-defined fix patterns. Recently, Liu et al. [141] used the fix patterns of FindBugs static violations [137] to fix semantic bugs. Concurrently, Ghanbari et al. [68] showed that straightforward application of fix patterns (i.e., mutators) on Java bytecode is effective for repair. They do not, however, provide a comprehensive assessment of the repair performance yielded by each implemented mutator.

Although the literature has reported promising results with fix patterns-based APR, to the best of our knowledge, no extensive assessment on the effectiveness of various patterns is performed. A few most recent approaches [91, 141, 144] reported which benchmark bugs are fixed by each of their patterns. Nevertheless, many relevant questions on the effectiveness of fix patterns remain unanswered.

This paper. Our work thoroughly investigates to what extent fix patterns are effective for program repair. In particular, emphasizing on the recurrence of some patterns in APR, we dissect their actual contribution to repair performance. Eventually, we explore three aspects of fix patterns:

- *Diversity*: How diverse are the fix patterns used by the state-of-the-art? We survey the literature to identify and summarize the available patterns with a clear taxonomy.
- *Repair performance*: How effective are the different patterns? In particular, we investigate the variety of real-world bugs that can be fixed, the dissection of repair results, and their tendency to yield plausible or correct patches.
- *Sensitivity to fault localization noise*: Are all fix patterns similarly sensitive to the false positives yielded by fault localization tools? We investigate sensitivity by assessing plausible patches as well as the suspiciousness rank of correctly-fixed bug locations.

Towards realizing this study, we implement an automated patch generation system, `TBar` (Template-Based automated program repair), with a super-set of fix patterns that are collected, summarized, curated and labeled from the literature data. We evaluate `TBar` on the Defects4J [98] benchmark, and provide the replication package in a public repository: `https://github.com/SerVal-DTF/TBar`.

Overall, our investigations have yielded the following findings:

1. **Record performance:** `TBar` creates a new higher baseline of repair performance: 74/101 bugs are correctly/plausibly fixed with perfect fault localization information and 43/81 bugs are fixed with realistic fault localization output, respectively.
2. **Fix pattern selection:** Most bugs are correctly fixed only by a single fix pattern while other patterns generate plausible patches. This implies that appropriate pattern prioritization can prevent from plausible/incorrect patches. Otherwise, APR tools might be overfitted in plausible but incorrect patches.

3. **Fix ingredient retrieval:** It is challenging for template-based APR to select appropriate donor code, which is an ingredient of patch generation when using fix patterns. Inappropriate donor code may cause plausible but incorrect patch generation. This motivates a new research direction: *donor code prioritization*.
4. **Fault localization noise:** It turns out that fault localization accuracy has a large impact on repair performance when using fix patterns in APR (e.g., applying a fix pattern to incorrect location yields plausible/incorrect patches).

7.2 Fix Patterns

For this study, we systematically review[1] the APR literature to identify approaches that leverage fix patterns. Concretely, we consider the program repair website[2], a bibliography survey of APR [174], proceedings of software engineering conference venues and journals as the source of relevant literature. We focus on approaches dealing with Java program bugs, and manually collect, from the paper descriptions as well as the associated artifacts, all pattern instances that are explicitly mentioned. Table 7.1 summarizes the identified relevant literature and the quantity of identified fix patterns targeting Java programs. Note that the techniques described in the last four papers (i.e., HDRepair, ssFix, CapGen, and SimFix papers) do not directly use fix patterns: they leverage code change operators or rules, which we consider similar to using fix patterns.

Table 7.1: Literature review on fix patterns for Java programs.

Authors	APR tool name	# of fix patterns	Publication Venue	Publication Year
Pan et al. [185]	-	27	EMSE	2009
Kim et al. [104]	PAR	10 (16*)	ICSE	2013
Martinez et al. [158]	jMutRepair	2	ISSTA	2016
Durieux et al. [56]	NPEfix	9	SANER	2017
Long et al. [147]	Genesis	3 (108*)	FSE	2017
D. Le et al. [122]	S3	4	FSE	2017
Saha et al. [199]	ELIXIR	8 (11*)	ASE	2017
Hua et al. [91]	SketchFix	6	ICSE	2018
Liu and Zhong [144]	SOFix	12	SANER	2018
Koyuncu et al. [113]	FixMiner	28	UL Tech Report	2018
Liu et al. [137]	-	174	TSE	2018
Rolim et al. [197]	REVISAR	9	UFERSA Tech Report	2018
Liu et al. [141]	AVATAR	13	SANER	2019
D. Le et al. [124]	HDRepair†	11	SANER	2016
Xin and Reiss [237]	ssFix†	34	ASE	2017
Wen et al. [226]	CapGen†	30	ICSE	2018
Jiang et al. [93]	SimFix†	16	ISSTA	2018

*In the PAR paper [104], 10 fix patterns are presented, but 16 fix patterns are released online[3]. In Genesis, 108 code transformation schemas are inferred for three kinds of defects. In ELIXIR, there is one fix pattern that consists of four sub-fix patterns.

7.2.1 Fix Patterns Inference

Fix patterns have been explored with the following four ways:

1. **Manual Summarization**: Pan et al. [185] identified 27 fix patterns from patches of five Java projects to characterize the fix ingredients of patches. They do not however apply the identified patterns to fix actual bugs. Motivated by this work, Kim et al. [104] summarized 10 fix patterns manually extracted from 62,656 human-written patches collected from Eclipse JDT.
2. **Mining**: Long et al. [147] proposed Genesis, to infer fix patterns for three kinds of defects from existing patches. Liu and Zhong [144] explored fix patterns from Q&A posts in Stack Overflow. Koyuncu et al. [113] mined fix patterns at the AST level from patches by using code change

[1]For conferences and journals, we consider ICSE, FSE, ASE, ISSTA, ICSME, SANER, TSE, TOSEM, and EMSE. The search keywords are 'program'+'repair', 'bug' +'fix'.
[2]`http://program-repair.org`
[3]`https://sites.google.com/site/autofixhkust/home/fix-templates`

differentiating tool [60]. Liu et al. [137] and Rolim et al. [197] proposed to mine fix patterns for static analysis violations. In general, mining approaches yield a large number of fix patterns, which are not always about addressing deviations in program behavior. For example, many patterns are about code style [141]. Recently, with *AVATAR* [141], we proposed an APR tool that considers static analysis violation fix patterns to fix semantic bugs.

3. **Pre-definition**: Durieux et al. [56] pre-defined 9 repair actions for null pointer exceptions by unifying the related fix patterns proposed in previous studies [55,103,151]. On the top of *PAR* [104], Saha et al. [199] further defined 3 new fix patterns to improve the repair performance. Hua et al. [91] proposed an APR tool with six pre-defined so-called code transformation schemas. We also consider operator mutations [158] as pre-defined fix patterns, as the number of operators and mutation possibilities is limited and pre-set. Xin and Reiss [237] proposed an approach to fixing bugs with 34 predefined code change rules at the AST level. Ten of the rules are not for transforming the buggy code but for the simple replacement of multi-statement code fragments. We discard these rules from our study to limit bias.
4. **Statistics**: Besides formatted fix patterns, researchers [93,226] also explored to automate program repair with code change instructions (at the abstract syntax tree level) that are statistically recurrent in existing patches [93,139,157,225,254]. The strategy is then to select the top-n most frequent code change instructions as fix ingredients to synthesize patches.

7.2.2 Fix Patterns Taxonomy

After manually assessing all fix patterns presented in the literature (cf. Table 7.1), we identified 15 categories of patterns labeled based on the code context (e.g., *a cast expression*), the code change actions (e.g., *insert an "if" statement with "instanceof" check*) as well as the targets (e.g., *ensure the program will no throw a ClassCastException.*). A given category may include one or several specialized sub-categories. Below, we present the labeled categories and provide the associated 35 **Code Change Patterns** described in simplified GNU diff pattern for easy understanding.

FP1. Insert Cast Checker. Inserting an *instanceof* check before one buggy statement if this statement contains at least one unchecked cast expression. **Implemented in:** PAR, Genesis, AVATAR, SOFix†, HDRepair†, SketchFix†, CapGen†, and SimFix†.

```
1  +  if (exp instanceof T) {
2        var = (T) exp; ......
3  +  }
```

where *exp* is an expression (e.g., a variable expression) and *T* is the casting type, while "……" means the subsequent statements dependent on the variable *var*. Note that, "†" denotes that the fix pattern is not specifically illustrated in the corresponding APR tools since the tools have some abstract fix patterns that can cover the fix pattern. The same notation applies to the following descriptions.

FP2. Insert Null Pointer Checker. Inserting a *null* check before a buggy statement if, in this statement, a field or an expression (of non-primitive data type) is accessed without a null pointer check. **Implemented in:** PAR, ELIXIR, NPEfix, Genesis, FixMiner, AVATAR, HDRepair†, SOFix†, SketchFix†, CapGen†, and SimFix†.

```
1  FP2.1: +  if (exp != null) {
2               ...exp...; ......
3         +  }
4  FP2.2: +  if (exp == null) return DEFAULT_VALUE;
5            ...exp...;
6  FP2.3: +  if (exp == null) exp = exp1;
7            ...exp...;
8  FP2.4: +  if (exp == null) continue;
```

```
9             ...exp...;
10 FP2.5: +   if (exp == null)
11        +      throw new IllegalArgumentException(...);
12            ...exp...;
```

where DEFAULT_VALUE is set based on the return type (RT) of the encompassing method as below:

$$\texttt{DEFAULT\_VALUE} = \begin{cases} \text{false}, & \text{if } RT = boolean; \\ 0, & \text{if } RT = primitive\ type; \\ \text{new String}(), & \text{if } RT = String; \\ \text{“return;”}, & \text{if } RT = void; \\ \text{null}, & otherwise. \end{cases} \tag{7.1}$$

exp1 is a compatible expression in the buggy program (i.e., that has the same data type as *exp*). **FP2.4** is specific to the case of a buggy statement within a loop (i.e., *for* or *while*).

FP3. Insert Range Checker. Inserting a range checker for the access of an array or collection if it is unchecked. **Implemented in:** PAR, ELIXIR, Genesis, SketchFix, AVATAR, SOFix† and SimFix†.

```
1         +   if (index < exp.length) {
2                 ...exp[index]...; ......
3         +   }
4   OR
5         +   if (index < exp.size()) {
6                 ...exp.get(index)...; ......
7         +   }
```

where *exp* is an expression representing an array or collection.

FP4. Insert Missed Statement. Inserting a missing statement before, or after, or surround a buggy statement. The statement is either an expression statement with a method invocation, or a *return/try-catch/if* statement. **Implemented in:** ELIXIR, HDRepair, SOFix, SketchFix, CapGen, FixMiner, and SimFix.

```
1  FP4.1: +   method(exp);
2  FP4.2: +   return DEFAULT_VALUE;
3  FP4.3: +   try {
4                 statement; ......
5         +   } catch (Exception e) {   ...   }
6  FP4.4: +   if (conditional_exp) {
7                 statement; ......
8         +   }
```

where *exp* is an expression from a buggy *statement*. It may be empty if the method does not take any argument. **FP4.4** excludes three fix patterns (**FP1**, **FP2**, and **FP3**) that are used with specific contexts.

FP5. Mutate Class Instance Creation. Replacing a class instance creation expression with a cast *super.clone()* method invocation if the class instance creation is in an overridden clone method. **Implemented in:** AVATAR.

```
1         public Object clone() {
2         -   ... new T();
3         +   ... (T) super.clone();
4         }
```

where *T* is the class name of the current class containing the buggy statement.

FP6. Mutate Conditional Expression. Mutating a conditional expression that returns a boolean value (i.e., `true` or `false`) by either updating it, or removing a sub conditional expression, or inserting a new conditional expression into it. **Implemented in:** PAR, ssFix, S3, HDRepair, ELIXIR, SketchFix, CapGen, SimFix, and AVATAR.

```
FP6.1: -  ...condExp1...
       +  ...condExp2...
FP6.2: -  ...condExp1 Op condExp2...
       +  ...condExp1...
FP6.3: -  ...condExp1...
       +  ...condExp1 Op condExp2...
```

where *condExp1* and *condExp2* are conditional expressions. *Op* is the logical operator '||' or '&&'. The mutation of operators in conditional expressions is not summarized in this fix pattern but in **FP11**.

FP7. Mutate Data Type. Replacing the data type in a variable declaration or a cast expression with another data type. **Implemented in:** PAR, ELIXIR, FixMiner, SOFix, CapGen, SimFix, AVATAR, and HDRepair†.

```
FP7.1: -  T1 var ...;
       +  T2 var ...;
FP7.2: -  ...(T1) exp...;
       +  ...(T2) exp...;
```

where both *T1* and *T2* denote two different data types. *exp* means the being casted expression (including variable).

FP8. Mutate Integer Division Operation. Mutating the integer division expressions to return a float value, by mutating its divisor or divider to make them be of type float. **Released by** Liu et al. [137], it is not implemented in any APR tool yet.

```
FP8.1: -  ...dividend / divisor...
       +  ...dividend / (double or float) divisor...
FP8.2: -  ...dividend / divisor...
       +  ...(double or float) dividend / divisor...
FP8.3: -  ...dividend / divisor...
       +  ...(1.0 / divisor) * dividend...
```

where *dividend* and *divisor* are integer number literals or integer-returned expressions (including variables).

FP9. Mutate Literal Expression. Mutating boolean, number, or String literals in a buggy statement with other relevant literals, or correspondingly-typed expressions. **Implemented in:** HDRepair, S3, FixMiner, SketchFix, CapGen, SimFix and ssFix†.

```
FP9.1: -  ...literal1...
       +  ...literal2...
FP9.2: -  ...literal1...
       +  ...exp...
```

where *literal1* and *literal2* are of the same type literals, but having different values (e.g., *literal1* is `true`, *literal2* is `false`). *exp* denotes any expression value of the same type as *literal1*.

FP10. Mutate Method Invocation Expression. Mutating the bu-ggy method invocation expression by adapting its method name or arguments. This pattern consists of four sub fix patterns:

1. Replacing the method name with another one which has a compatible return type and same parameter type(s) as the buggy method that was invoked.
2. Replacing at least one argument with another expression which has a compatible data type. Replacing a literal or variable is not included in this fix pattern, but rather in **FP9** and **FP13** respectively.
3. Removing argument(s) if the method invocation has the suitable overridden methods.
4. Inserting argument(s) if the method invocation has the suitable overridden methods.

Implemented in: PAR, HDRepair, ssFix, ELIXIR, FixMiner, SOFix, SketchFix, CapGen, and SimFix.

```
FP10.1: -  ...method1(args)...
        +  ...method2(args)...
FP10.2: -  ...method1(arg1, arg2, ...)...
        +  ...method1(arg1, arg3, ...)...
FP10.3: -  ...method1(arg1, arg2, ...)...
        + ... method1(arg1, ...)...
FP10.4: -  ...method1(arg1, ...)...
        +  ...method1(arg1, arg2, ...)...
```

where *method1* and *method2* are the names of invoked methods. *args*, *arg1*, *arg2* and *arg3* denote the argument expressions in the method invocation. Note that, code changes on class instance creation, constructor and super constructor expressions are also included in these four fix patterns.

FP11. Mutate Operators. Mutating an operation expression by mutating its operator(s). We divide this fix pattern into three sub-fix patterns following the operator types and mutation actions.

1. Replacing one operator with another operator from the same operator class (e.g., relational or arithmetic).
2. Changing the priority of arithmetic operators.
3. Replacing `instanceof` operator with (in)equality operators.

Implemented in: HDRepair, ssFix, ELIXIR, S3, jMutRepair, SOFix, FixMiner, SketchFix, CapGen, SimFix, AVATAR, and PAR†.

```
FP11.1: -  ...exp1 Op1 exp2...
        +  ...exp1 Op2 exp2...
FP11.2: -  ...(exp1 Op1 exp2) Op2 exp3...
        +  ...exp1 Op1 (exp2 Op2 exp3)...
FP11.3: -  ...exp instanceof T...
        +  ...exp != null...
```

where *exp* denotes the expressions in the operation and *Op* is the associated operator.

FP12. Mutate Return Statement. Replacing the expression (excluding literals, variables, and conditional expressions) in a return statement with a compatible expression. **Implemented in:** ELIXIR, SketchFix, and HDRepair†.

```
        -   return exp1;
        +   return exp2;
```

where *exp1* and *exp2* represent the returned expressions.

FP13. Mutate Variable. Replacing a variable in a buggy statement with a compatible expression (including variables and literals). **Implemented in:** S3, SOFix, FixMiner, SketchFix, CapGen, SimFix, AVATAR, and ssFix†.

```
FP13.1: -  ...var1...
        +  ...var2...
FP13.2: -  ...var1...
        +  ...exp...
```

where *var1* denotes a variable in the buggy statement. *var2* and *exp* represent respectively a compatible variable and expression of the same type as *var1*.

FP14. Move Statement. Moving a buggy statement to a new position. **Implemented in:** PAR.

```
        -   statement;
            ......
        +   statement;
```

where *statement* represents the buggy statement.

FP15. Remove Buggy Statement. Deleting entirely the buggy statement from the program. **Implemented in:** HDRepair, SOFix, FixMiner, CapGen, and AVATAR.

```
FP15.1:      ......
        -   statement;
            ......
FP15.2: -  methodDeclaration(Arguments) {
        -   ......; statement;......
        -  }
```

where *statement* denotes any identified buggy statement, and *method* represents the encompassing method.

7.2.3 Analysis of Collected Patterns

We provide a study of the collected fix patterns following quantitative (overall set) and qualitative (per fix pattern) aspects. Table 7.2 assesses the fix patterns in terms of four qualitative dimensions:

1. **Change Action**: what high-level operations are applied on a buggy code entity? On the one hand, *Update* operations replace the buggy code entity with retrieved donor code, while *Delete* operations just remove the buggy code entity from the program. On the other hand, *Insert* operations insert an otherwise missing code entity into the program, and *Move* operations change the position of the buggy code entity to a more suitable location in the program.
2. **Change Granularity**: what kinds of code entities are directly impacted by the change actions? This entity can be an entire *Method*, a whole *Statement* or specifically targeting an *Expression* within a statement.
3. **Bug Context**: what specific AST nodes of code entities are used to match fix patterns.
4. **Change Spread**: the number of statements impacted by each fix pattern.

Quantitatively, as summarized in Table 7.3, 17 fix patterns are related to *Update* change actions, 4 fix patterns implement *Delete* actions, 13 fix patterns *Insert* extra code, and only 1 fix pattern is associated to *Move* change action.

In terms of change granularity, 21 and 17 fix patterns are applied respectively at the expression and statement code entity levels[4]. Only 1 fix pattern is suitable at the method level.

[4]Among these, four sub-fix patterns (**FP10**) can be applied to either expressions or statements, given that constructor and super-constructor code entities in Java program are grouped into statement level in terms of abstract syntax tree by Eclipse JDT.

Table 7.2: Change properties of fix patterns.

Fix Pattern	Change Action	Change Graunlarity	Bug Context	Change Spread
FP1	Insert	statement	cast expression	single
FP2.1	Insert	statement	a variable or an expression returning non-primitive-type data	single
FP2.(2,3,4,5)				dual
FP3	Insert	statement	element access of array or collection variable	single
FP4.(1,2,3,4)	Insert	statement	any statement	single
FP5	Update	expression	class instance creation expression and clone method	single
FP6.1	Update	expression	conditional expression	single
FP6.2	Delete			
FP6.3	Insert			
FP7.1	Update	expression	variable declaration expression	single
FP7.2	Update	expression	cast expression	single
FP8.(1,2,3)	Update	expression	integral division expression	single
FP9.(1,2)	Update	expression	literal expression	single
FP10.1	Update	expression, or statement	method invocation, class instance creation, constructor, or super constructor	single
FP10.2				
FP10.3	Delete			
FP10.4	Insert			
FP11.1	Update	expression	assignment or infix-expression	single
FP11.2	Update	expression	arithmetic infix-expression	single
FP11.3	Update	expression	instance of expression	single
FP12	Update	expression	return statement	single
FP13.(1, 2)	Update	expression	variable expression	single
FP14	Move	statement	any statement	single or multiple
FP15.1	Delete	statement	any statement	single or multiple
FP15.2	Delete	method	any statement	multiple

Table 7.3: Diversity of fix patterns w.r.t change properties.

Action Type	# fix patterns	Granularity	# fix patterns	Spread	# fix patterns
Update	17	Expression	21	Single-Statement	30
Delete	4	Statement	17		
Insert	13	Method	1	Multiple-Statements	7
Move	1				

Overall, we note that 30 fix patterns are applicable to a single statement, while 7 fix patterns can mutate multiple statements at the same time. Among these patterns, **FP14** and **FP15.1** can both mutate single and multiple statements.

7.3 Setup for Repair Experiments

In order to assess the effectiveness of fix patterns in the taxonomy presented in Section 7.2, we design program repair experiments using the fix patterns as the main ingredients. The produced APR system is then assessed on a widely-used benchmark in the repair community to allow reliable comparison against the state-of-the-art.

7.3.1 TBar: A Baseline APR System

Based on the investigations of recurrently-used fix patterns, we build TBar, a template-based APR tool which integrates the 35 fix patterns presented in Section 7.2. We expect the APR community to consider TBar as a baseline APR tool: new approaches must come up with novel techniques for solving auxiliary issues (e.g., repair precision, search space optimization, fault locations re-prioritization, etc.) to boost automated program repair beyond the performance that a straightforward application of common fix patterns can offer. Figure 7.1 overviews the workflow that we have implemented in TBar. We describe in the following subsections the role and operation of each process as well as all necessary implementation details.

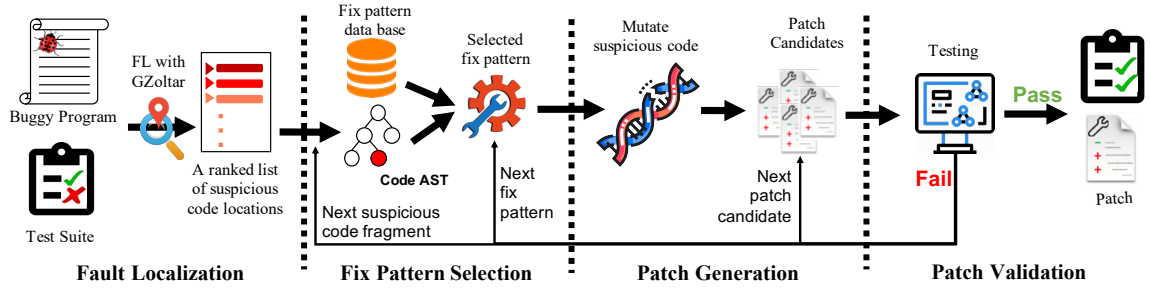


Figure 7.1: The overall workflow of TBar.

7.3.1.1 Fault Localization

Fault localization is necessary for template-based APR as it allows to identify a list of suspicious code locations (i.e., buggy statements) on which to apply the fix patterns. TBar leverages the GZoltar [40] framework to automate the execution of test cases for each buggy program. In this framework, we use the Ochiai [3] ranking metric to compute the suspiciousness scores of statements that are likely to be the faulty code locations. This ranking metric has been demonstrated in several empirical studies [187, 208, 236, 241] to be effective for localizing faults in object-oriented programs. The GZoltar framework for fault localization is also widely used in the literature of APR [93, 113, 140, 141, 143, 158, 226, 237, 239, 240], allowing for a fair assessment of TBar's performance against the state-of-the-art.

7.3.1.2 Fix Pattern Selection

In the execution of the repair pipeline, once the fault localization process yields a list of suspicious code locations, TBar sequentially attempts to select the encoded fix patterns from its database of fix patterns for each statement in the locations list. The selection of fix patterns is conducted in a naïve way based on the AST context information of each suspicious statement. Specifically, TBar sequentially traverses each node of the suspicious statement AST from its first child node to its last leaf node and tries to match each node against the context AST of the fix pattern. If a node can match any bug context presented in Table 7.2, a related fix pattern will be matched to generate patch candidates with the corresponding code change pattern. If the node is not a leaf node, TBar keeps traversing its children nodes. For example, if the first child node of a suspicious statement is a method invocation expression, it will be first matched with **FP10. Mutate Method Invocation Expression** fix pattern. If the children nodes of the method invocation start from a variable reference, it will be matched with **FP13. Mutate Variable** fix pattern as well. Other fix patterns follow the same manner. After all expression nodes of a suspicious statement are matched with fix patterns, TBar further matches fix patterns from statement and method levels respectively.

7.3.1.3 Patch Generation and Validation

When a matching fix pattern is found (i.e., a pattern is selected for a suspicious statement), a patch is generated by mutating the statement, then the patched program is run against the test suite. If the patched program passes all tests successfully, the patch candidate is considered as a *plausible* patch [194]. Once such a plausible patch is identified, `TBar` stops generating other patch candidates for this bug to fix bugs in a standard and practical program repair workflow [140, 141, 158, 239, 240], but does not generate all plausible patches for each bug, unlike PraPR [68]. Otherwise, the pattern selection and patch generation process is resumed until all AST nodes of buggy code are traversed. When several fix pattern contexts match one node, their actions are used for ordering: `TBar` prioritizes Update over Insert that is over Delete, which is prioritized over Move. In case of multiple donor code options for a given fix pattern, the candidate patches (each generated with a specific donor code) are ordered based on the distances between donor code node and buggy code node in the AST of the buggy code file: priority is given to smaller distances. Due to space limitation, detailed steps, illustrated in an algorithmic pseudo-code, are released in the replication package.

Considering that some buggy programs have several buggy locations, if a patch candidate can make a buggy program pass a sub-set of previously failing test cases without failing any previously passing test cases, this patch is considered as a plausible sub-patch of this buggy program. `TBar` will further validate other patch candidates, until either a plausible patch is generated, or all patch candidates are validated, or `TBar` exhausts the time limitation set (i.e., three hours) for repair attempts.

If a plausible patch is generated, we further manually check the equivalence between this patch and the ground-truth patch provided by developers and available in the Defects4J benchmark. If the plausible patch is semantically equivalent to the ground-truth patch, the plausible patch is considered as *correct*. Otherwise, it is only considered as plausible. We offer a replication package with extensive details on pattern implementation within `TBar`. Source code is publicly available in the aforementioned GitHub repository.

7.3.2 Assessment Benchmark

For our empirical assessments, we selected the Defects4J [98] dataset as the evaluation benchmark of `TBar`. This benchmark includes test cases for buggy Java programs with the associated developer fixes. Defects4J is an ideal benchmark for the objective of this study, since it has been widely used by most recent state-of-the-art APR systems targeting Java program bugs. Table 7.4 provides summary statistics on the bugs and test cases available in the version 1.2.0[5] of Defects4J which we use in this study.

Table 7.4: Defects4J dataset information.

Project	Chart (C)	Closure (Cl)	Lang (L)	Math (M)	Mockito (Mc)	Time (T)	Total
# bugs	26	133	65	106	38	27	395
# test cases	2,205	7,927	2,245	3,602	1,457	4,130	21,566
# fixed bugs by all APR tools (cf. [140, 141])	13	16	28	37	3	4	101

Overall, we note that, to date, **101** Defects4J bugs have been correctly fixed by at least one APR tool published in the literature. Nevertheless, we recall that SimFix [93] currently holds the record number of bugs fixed by a single tool, which is **34**.

[5] `https://github.com/rjust/defects4j/releases/tag/v1.2.0`

7.4 Assessment

This section presents and discusses the results of repair experiments with `TBar`. In particular, we conduct two experiments for:

- **Experiment #1:** Assessing the effectiveness of the various fix patterns implemented in `TBar`. To avoid the bias that fault localization can introduce with its false positives (cf. [140]), we directly provide perfect localization information to `TBar`.
- **Experiment #2:** Evaluating `TBar` in a normal program repair scenario. We investigate in particular the tendency of fix patterns to produce more or less incorrect patches.

7.4.1 Repair Suitability of Fix Patterns

Our first experiment focuses on assessing the patch generation performance of fix patterns for real bugs. In particular, we investigate three research questions in Experiment #1.

Research Questions for Experiment #1

RQ1. How many real bugs from Defects4J can be correctly fixed by fix patterns from our taxonomy?
RQ2. Can each Defects4J bug be fixed by different fix patterns?
RQ3. What are the properties of fix patterns that are successfully used to fix bugs?

In a recent study, Liu et al. [140] reported how fault localization techniques substantially affect the repair performance of APR tools. Given that, in this experiment, the APR tool (namely `TBar`) is only used as a means to apply the fix patterns in order to assess their effectiveness, we must eliminate the fault localization bias. Therefore, we assume that the bug positions at statement level are known, and we directly provide it to the patch generation step of `TBar`, without running any fault localization tool (which is part of the normal APR workflow, see Figure 7.1). To ensure readability across our experiments, we denote this version of the APR system as $\texttt{TBar}_p$ (where p stands for *perfect localization*). Table 7.5 summarizes the experimental results of $\texttt{TBar}_p$.

Table 7.5: Number of bugs fixed by fix patterns with $TBar_p$.

Fixed Bugs	C	Cl	L	M	Mc	T	Total
# of Fully Fixed Bugs	12/13	20/26	13/18	23/35	3/3	3/6	74/101
# of Partially Fixed Bugs	2/4	3/6	1/4	0/4	0/0	1/1	7/20

*We provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs fixed with plausible patches. The same notation applies to Table 7.7.

Among 395 bugs in the Defects4J benchmark, $\texttt{TBar}_p$ can generate plausible patches for 101 bugs. 74 of these bugs are fixed with correct patches. We also note that $\texttt{TBar}_p$ can partially fix[6] 20 bugs with plausible patches, and 8 of them are correct. In a previous study, the `kPAR` [140] baseline tool (i.e., a Java implementation of the PAR [104] seminal template-based APR tool) was correctly/plausibly fixing 36/55 Defects4J bugs when assuming perfect localization.

While the results of $\texttt{TBar}_p$ are promising, ∼79%(=314/395) of bugs cannot be correctly fixed with the available fix patterns. We manually investigated these unfixed bugs and make the following observations as research directions for improving the fix rates:

1. *Insufficient fix patterns.* Many bugs are not fixed by $\texttt{TBar}_p$ simply due to the absence of matching fix patterns. This suggests that the fix patterns collected in the literature are far from being representative for real-world bugs. The community must thus keep contributing with effective techniques for mining fix patterns from existing patches.

[6]Partial fix: a patch makes the buggy program pass a part of previously failed test cases without causing any new failed test cases [140].

2. *Ineffective search of fix ingredients.* Template-based APR is a kind of search-based APR [226]: some fix patterns require donor code (i.e., fix ingredients) to generate actual patches. For example, as shown in Figure 7.2, to apply the relevant fix pattern **FP9.2**, one needs to identify fix ingredient "`ImageMapUtilities.htmlEs- cape`" as the necessary in generating the patch. The current implementation of `TBar` limits its search space for donor code to the "***local***" file where the bug is localized. It is a limitation to find the correct donor code, but it reduces the risk of search space explosion. In addition, `TBar` leverages the context of buggy code to prune away irrelevant fix ingredients. Therefore, some bugs cannot be fixed by `TBar` although its fix pattern can match with code change actions. With more effective search strategies (e.g., larger search space such as fix ingredients from other projects as in [143]), there might be more chances to fix more bugs.

```
1    public String generateToolTipFragment(String toolTipText) {
2 -      return " title=\"" + toolTipText
3 +      return " title=\"" + ImageMapUtilities.htmlEscape(toolTipText)
4                + "\" alt=\"\"";
5    }
6    Code Change Action:
7    Replace variable "toolTipText" with a method invocation expression "
      ImageMapUtilities.htmlEscape(toolTipText)".
8    Matchable fix pattern:  FP9.2.
```

Figure 7.2: Patch and code change action of fixing bug C-10.

> ***RQ1:*** *The collected fix patterns can be used to correctly fix 74 real bugs from the Defects4J dataset. A larger portion of the dataset remains however unfixed by $\texttt{TBar}_p$, notably due to (1) the limitations of the fix patterns set and to (2) the naïve search strategy for finding relevant fix ingredients to build concrete patches from patterns.*

Figure 7.3 summarizes the statistics on the number of bugs that can be fixed by one or several fix patterns. The *Y*-axis denotes the number of fix patterns (i.e., $n = 1, 2, 3, 4, 5$, and >5) that can generate *plausible* patches for a number of bugs (*X*-axis). The legend indicates that "**P**" represents the number of plausible patches generated by $\texttt{TBar}_p$ (i.e., those that are not found to be correct). "#k", where $k \in [1, 4]$, indicates that a bug can be correctly fixed by only k fix patterns (although it may be plausibly fixed by more fix patterns).

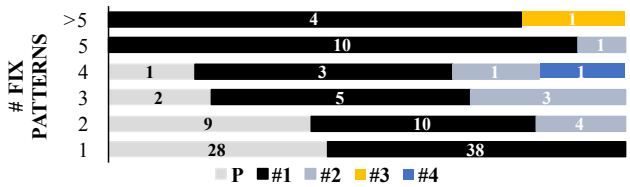


Figure 7.3: Number of bugs plausibly and correctly fixed by single or multiple fix patterns.

Consider for the bottom-most bar in Figure 7.3: 66 (=28+38) bugs can be plausibly fixed by a single pattern (*Y*-axis value is 1); it turns out that only 38 of them are correctly fixed. Note that several patterns can generate (plausible) patches for a bug, but not all patches are necessarily correct. For example, in the case of the top-most bar in Figure 7.3, 5 bugs are each plausibly fixed by over 5 fix patterns. However, only 1 bug is correctly fixed by 3 fix patterns.

In summary, 86% ($=\frac{38+10+5+3+10+4}{74+7}$) of correctly fixed bugs (74 fully and 7 partially fixed bugs) are exclusively fixed correctly by single patterns. In other words, generally, several fix patterns can generate patches that can pass all test cases but, in most cases, the bug is correctly fixed by only one pattern. This finding suggests that it is necessary to carefully select an appropriate fix pattern when attempting to fix a bug, in order to avoid plausible patches which may prevent the discovery of correct patches by halting the repair process (given that all tests are passing on the plausible patch).

> ***RQ2:*** *Some bugs can be plausibly fixed by different fix patterns. However, in most cases, only one fix pattern is adequate for generating a correct patch. This finding suggests a need for new research on fix pattern prioritization.*

Table 7.6 details which bug is fixed by which fix pattern(s). We note that five fix patterns (i.e., FP3, FP4.3, FP5, FP7.2 and FP11.3) cannot be used to generate a plausible patch for any Defects4J bug. Two fix patterns (i.e., FP9.2 and FP12) lead to plausible patches for some bugs, but none of them is correct. It does not necessarily suggest that the aforementioned fix patterns are useless (or ineffective) in APR. Instead, two reasons can explain their performance:

- The search for donor code may be inefficient for finding relevant ingredients for applying these patterns
- The Defects4J dataset does not contain the types of bugs that can be addressed by these fix patterns.

Table 7.6: Defects4j bugs fixed by fix patterns.

Bug ID	FP1	FP2					FP3	FP4				FP5	FP6			FP7		FP8			FP9		FP10				FP11			FP12	FP13		FP14	FP15		
		1	2	3	4	5		1	2	3	4		1	2	3	1	2	1	2	3	1	2	1	2	3	4	1	2	3		1	2		1	2	
C-1															❍												●					❍	❍	❍			1/5
C-4		●	❍		●																															2/3
C-7											❍																				◐					1/2
C-8																															●					1/1
C-9															●							❍														1/2
C-11																															●					1/1
C-12								●																												1/1
C-14			●	●																											❍					2/3
C-18								❍							❍												❍				❍				●	1/5
C-19					●																															1/1
C-20																							●													1/1
C-24																															●					1/1
C-25			●								❍																				❍					1/3
C-26		●	❍	●																																2/3
Cl-2		●	❍	❍																																1/3
Cl-4																						●														1/1
Cl-6														❍	❍												❍				❍	❍			●	1/6
Cl-10																						●														1/1
Cl-11													❍		❍												❍				❍				●	1/5
Cl-13																																	●			1/1
Cl-18														●													❍									1/2
Cl-21													❍		❍												❍				❍				●	1/5
Cl-22													❍		❍												❍				❍				●	1/5
Cl-31														●													❍									1/2
Cl-38													❍	❍													●									1/3
Cl-40																					●															1/1
Cl-46																																			●	1/1
Cl-62													❍	❍	❍												◐								❍	1/5
Cl-63													❍	❍	❍												◐								❍	1/5
Cl-70																					●															1/1
Cl-73																											●									1/1
Cl-85																																				1/1
Cl-86																					●															1/1
Cl-102								●																									●			2/2
Cl-106																																❍				1/2
Cl-115													❍		❍																❍		❍			1/5
Cl-126			❍	❍									❍		❍																❍					1/6
L-6																															●					1/1
L-7													❍		❍																❍					1/4
L-10														◐																						2/2
L-15													❍	●	❍												❍								❍	1/5
L-22														❍	❍												◐				❍				❍	1/5
L-24															●																					1/1
L-26																										●										1/1
L-33		●																																		1/1
L-39		●									❍																					❍				1/3
L-47				●																																1/1
L-51									●																											1/1
L-57		❍	❍																											❍	●	●				2/5
L-59																															●					1/1
L-63															❍								❍				❍				❍	❍	❍	●		1/7
M-4			●																																	1/1
M-5																															●	❍				1/2
M-11																		●	●	●	●															4/4
M-15															◐																					1/1
M-22																					●	❍														1/2
M-30																	◐																			1/1
M-33															❍							❍									●					1/3
M-34																																●				1/1
M-35								●																												1/1
M-50											❍		❍		❍							❍						❍	❍		❍	❍		●		1/9
M-57																	●																			1/1
M-58																								●												1/1
M-59																															●	❍				1/2
M-65																											●									1/1
M-70																									●											1/1
M-75																						●														1/1
M-77											❍																●				❍				●	2/4
M-79																	◐																			1/1
M-80																						❍						●			❍	❍				1/4
M-82											❍				❍							❍					●				❍					1/5
M-85											❍		◐		◐							❍					●				❍	❍		❍		3/8
M-89	●																																			1/1
M-98																															●					1/1
Mc-26																					●															1/1
Mc-29		●	●																																	2/2
Mc-38			●	●																																2/2
T-3											◐																									1/1
T-7																															●	❍				1/2
T-19															❍												●									1/2
T-26																									●											1/1
# 1	1	6	5	4	1	1	0	3	1	0	1	0	1	3	5	3	0	1	1	1	6	0	3	1	1	3	11	1	0	0	12	2	2	13	2	
# 2	1	7	10	6	1	1	0	4	1	0	14	0	15	12	32	3	0	1	1	1	6	7	4	2	2	3	24	2	0	1	43	19	6	25	4	

∗ ● indicates that the bug is correctly fixed and ❍ indicates that the generated patch is plausible but not correct. ◐means that the fix pattern can generate both correct patch and plausible patch for a bug. ● and ❍ denote that the bug can be partially fixed by the corresponding fix pattern. In the last column, we provide x/y numbers: x is the number of fix patterns that can generate correct patches for a bug, and y is the number of fix patterns that can generate plausible patches for a bug. **Note that**, the bugs that can be plausible but incorrectly fixed by fix patterns are not shown in this table. **# 1**: number of bugs correctly fixed by a fix pattern. **# 2**: number of bugs plausible fixed by a fix pattern.

In addition, twenty (20) fix patterns lead to the generation of correct patches for some bugs. Most of these fix patterns are involved in the generation of plausible patches (which turn out to be incorrect).

Interestingly, we found the cases of six (6) fix patterns which can generate several[7] patch candidates, some which being correct and others being only plausible, for the same 10 bugs (as indicated in Table 7.6 with '◐'). This observation further highlights the importance of selecting a relevant donor code for synthesizing patches: selecting an inappropriate donor code can lead to the generation of a plausible (but incorrect) patch, which will impede the generation of correct patches in a typical repair pipeline.

> *Aside from fix patterns, fix ingredients collected in donor code are essential to be properly selected to avoid patches that are plausible but may yet be incorrect.*

We further inspect properties of fix patterns, such as change actions, granularity, and the number of changed statements in patches. The statistics are shown in Figure 7.4, highlighting the number of plausible (but incorrect) and correct patches for the different property dimensions through which fix patterns can be categorized.

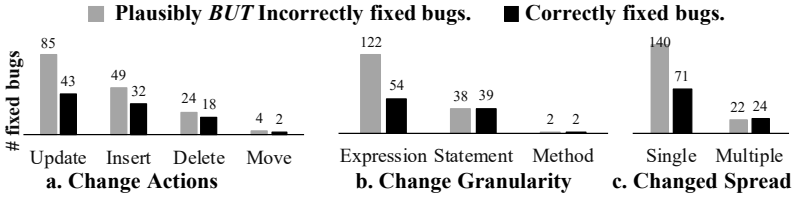


Figure 7.4: Qualitative statistics of bugs fixed by fix patterns.

More bugs are fixed by *Update* change actions than any by any other actions. Similarly, fix patterns targeting expressions fix more bugs correctly than patterns targeting statements and methods. However, fix patterns mutating whole statements have a higher rate of correct patches among their plausible generated patches. Finally, fix patterns changing only single statements can correctly fix more bugs than those touching multiple statements. Fix patterns targeting multi-statements have however a higher rate of correctness.

> ***RQ3:*** *There are noticeable differences between successful repair among fix patterns depending on their properties related to implemented change actions, change granularity and change spread.*

7.4.2 Repair Performance Comparison: `TBar` vs State-of-the-art APR Tools

Our second experiment evaluates `TBar` in a realistic setting for patch generation, allowing for reliable comparison against the state-of-the-art in the literature. Concretely, we investigate two research questions in Experiment #2.

Research Questions for Experiment #2

RQ4. What performance can be achieved by `TBar` in a standard and practical repair scenario?

RQ5. To what extent are the different fix patterns sensitive to noise in fault localization (i.e., spotting buggy code locations)?

In this experiment we implement a realistic scenario, using a normal fault localization (i.e., no assumption of perfect localization as for $\texttt{TBar}_p$) on Defects4J bugs. To enable a fair comparison with performance results recorded in the literature, `TBar` leverages a standard configuration in the literature [140] with GZoltar [40] and Ochiai [3]. Furthermore, `TBar` does not utilize any additional technique to improve the accuracy of fault localization, such as crashed stack trace (used by ssFix [237]), predicate switching [251] (used by ACS [239]), or test case purification [242] (used by SimFix [93]).

[7] Note that, in this experiment $\texttt{TBar}_p$ generates and assesses all possible patch candidates for a given pair "bug location - fix pattern" with varying ingredients.

With respect to the patch generation step, contrary to the experiment with $\texttt{TBar}_p$ where all positions of multi-locations bugs were known (cf. Section 7.4.1), TBar adapts a "first-generated and first-selected" strategy to progressively apply fix patterns, one at a time, in various suspicious code locations: TBar generates a patch p_i, using a fix pattern that matches a given bug. If p_i passes a subset of previously-failing test cases without failing any previously-passing test case, TBar selects p_i as a plausible patch for the bug. Then, TBar continues to validate another patch p_{i+1} (which can be generated by the same fix pattern on the same code entity with other ingredients, or on another code location). When p_{i+1} passes a subset of test cases as p_i, if p_{i+1} is generated for the same buggy code entity as p_i, p_{i+1} will be abandoned; otherwise, TBar takes p_{i+1} as another plausible patch as well. Through this process, TBar creates a patch set $P = \{ p_i, p_{i+1}, ...\}$ of plausible patches. Here, as soon as any patch can pass all the given test cases for a given bug, TBar takes it as a plausible patch for the given bug, which is regarded as a *fully-fixed* bug, and all $p_i \in P$ will be abandoned. Otherwise, our tool yields P, a set of plausible patches that can each partially fix the given bug.

We run the TBar APR system against the buggy programs of the Defects4J dataset. Table 7.7 presents the performance of TBar in comparison with recent state-of-the-art APR tools from the literature. TBar can fix 81 bugs with plausible patches, 43 of which are correctly fixed. No other APR tool had reached this number of fixed bugs. Nevertheless, its precision (ratio of correct vs. plausible patches) is lower than some recent tools such as CapGen and SimFix which employs sophisticated techniques to select fix ingredients. Nonetheless, it is noteworthy that, despite using fix patterns catalogued in the literature, we can fix three bugs (namely Cl-86,L-47,M-11) which had never been fixed by any APR system: M-11 is fixed by a pattern found by a standalone fix pattern mining tool [137] but which was not encoded by any APR system yet. Cl-86 and L-47 are fixed by patterns that were not applied to Defects4J.

Table 7.7: Comparing TBar against the state-of-the-art APR tools.

Project	jGenProg	jKali	jMutRepair	HDRepair	Nopol	ACS	ELIXIR	JAID	ssFix	CapGen	SketchFix	FixMiner	LSRepair	SimFix	kPAR	AVATAR	**TBar**	
																	Fully fixed	Partially fixed
Chart	0/7	0/6	1/4	0/2	1/6	2/2	4/7	2/4	3/7	4/4	6/8	5/8	3/8	4/8	3/10	5/12	**9/14**	0/4
Closure	0/0	0/0	0/0	0/7	0/0	0/0	0/0	5/11	2/11	0/0	3/5	5/5	0/0	6/8	5/9	8/12	**8/12**	1/5
Lang	0/0	0/0	0/1	2/6	3/7	3/4	8/12	1/8	5/12	5/5	3/4	2/3	8/14	**9**/13	1/8	5/11	5/**14**	0/3
Math	5/18	1/14	2/11	4/7	1/21	12/16	12/19	1/8	10/26	12/16	7/8	12/14	7/14	14/26	7/18	6/13	**19/36**	0/4
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/2	**2/2**	1/2	0/0
Time	0/2	0/2	0/1	0/1	0/1	1/1	**2**/3	0/0	0/**4**	0/0	0/1	1/1	0/0	1/1	1/2	1/3	1/3	1/2
Total	5/27	1/22	3/17	6/23	5/35	18/23	26/41	9/31	20/60	21/25	19/26	25/31	19/37	34/56	18/49	27/53	**43/81**	2/18
P(%)	18.5	4.5	17.6	26.1	14.3	78.3	63.4	29.0	33.3	**84.0**	73.1	80.6	51.4	60.7	36.7	50.9	53.1	11.1

*"**P**" is the probability of generated plausible patches to be correct. The data of other APR tools are excerpted from the corresponding work. **kPAR** [140] is an open-source implementation of PAR [104].

> ***RQ4:*** *TBar outperforms all recent state-of-the-art APR tools that were evaluated on the Defects4J dataset. It correctly fixes 43 bugs, while the runner-up (SimFix) is reported to correctly fix 34 bugs.*

It is noteworthy that TBar performs significantly less than $\texttt{TBar}_p$ (43 vs. 74 correctly fixed bugs). This result is in line with a recent study [140], which demonstrated that fault localization imprecision is detrimental to APR repair performance. Table 7.6 summarizes information about the number of bugs each fix pattern contributed to fixing with $\texttt{TBar}_p$. While only 4 fix patterns did not lead to the generation of any plausible patch when assuming perfect localization. With TBar, it is the case for 13 fix patterns (see Table 7.8). This observation further confirms the impact of fault localization noise.

Table 7.8: Per-pattern repair performance.

	FP1	FP2					FP3	FP4				FP5	FP6			FP7		FP8			FP9		FP10				FP11			FP12	FP13		FP14	FP15	
		1	2	3	4	5		1	2	3	4		1	2	3	1	2	1	2	3	1	2	1	2	3	4	1	2	3		1	2		1	2
Correct	1	4	2	1	0	1	0	1	0	0	0	0	0	0	3	3	0	0	0	1	2	0	1	1	1	1	7	1	0	0	9	1	0	2	2
Avg position*	(1)	(16)	(1)	(5)	-	(5)	-	(5)	-	-	-	-	-	-	(23)	(16)	-	-	-	(9)	(1)	-	(2)	(62)	(6)	(1)	(12)	(18)	-	-	(5)	(1)	-	(2)	(1)
Plausible (all)	1	7	4	1	0	1	0	3	0	0	0	0	1	0	11	4	0	0	0	1	4	0	2	2	1	1	12	1	0	0	25	4	1	7	5
Avg position*	(1)	(12)†	(191)	(5)	-	(5)	-	(20)	-	-	-	-	(8)	-	(27)†	(15)	-	-	-	(9)	(18)	-	(4)	(49)	(6)	(1)	(15)†	(18)	-	-	(8)†	(20)	(15)	(26)	(16)

*Average position of the exact buggy position in the list of suspicious statements yield by fault localization tool. † The exact buggy positions of some bugs cannot be yield by fault localizaiton tool.

We propose to examine the locations where TBar applied fix patterns to generate plausible but incorrect patches. As shown in Figure 7.5, TBar has made changes on incorrect positions (i.e., non-buggy locations) for 24 out of the 38 fully-fixed and 15 out of the 16 partially-fixed bugs.

Even when TBar applies a fix pattern to the precise buggy location, the generated patch may be

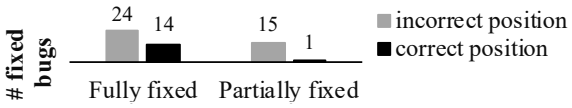
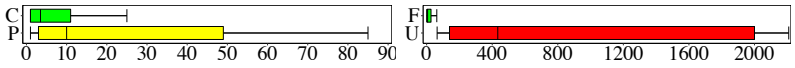


Figure 7.5: The mutated code positions of plausibly but incorrectly fixed bugs.

incorrect. As shown in Figure 7.5, 14 patches that fully fix Defects4J bugs mutate the correct locations: in 3 cases, the fix patterns were inappropriate; in 2 other cases, `TBar` failed to locate relevant donor code; for the remaining, `TBar` does not support the required fix patterns.

Finally, Figure 7.6 illustrates the impact of fault localization performance: unfixed bugs (but correctly fixed by $\texttt{TBar}_p$) are generally more poorly localized than correctly fixed bugs. Similarly, we note that many plausible but incorrect patches are generated for bugs which are not well localized (i.e., several false positive buggy locations are mutated leading to plausible but incorrect patches).



*** X-axis: Bug positions in suspicious list reported by fault localization.**

Figure 7.6: Distribution of the positions of buggy code locations in fault localization list of suspicious statements. *C* and *P* denote Correctly- and Plausibly- (but incorrectly) fixed bugs, respectively. *F* and *U* denote Fixed and Unfixed bugs.

Average positions bugs (in fault localization suspicious list) are also provided in Table 7.8. It appears that some fix patterns (e.g., FP2.1, FP6.3, FP10.2) can correctly fix bugs that are poorly localized, showing less sensitivity to fault localization noise than others.

> ***RQ5:*** *Fault localization noise has a significant impact on the performance of `TBar`. Fix patterns are diversely sensitive to the false positive locations that are recommended as buggy positions.*

7.5 Discussion

Overall, our investigations reveal that a large catalogue of fix patterns can help improve APR performance. However, at the same time, there are other challenges that must be dealt with: more accurate fault localization, effective search of relevant donor code, fix pattern prioritization. While we will work on some of these research directions in future work, we discuss in this section some threats to validity of the study and practical limitations of `TBar`.

7.5.1 Threats to Validity

Threats to external validity include the target language of this study, i.e., Java. Fix patterns studied in this paper only cover the fix patterns targeting at Java program bugs released by the state-of-the-art pattern-based APR systems. However, we believe that most fix patterns presented in this study could be applied to other languages since fix patterns are illustrated as abstract syntax tree level. Another threat to external validity could be the fix pattern diversity. Our study may not consider all available fix patterns so far in the literature. To reduce this threat, we systematically reviewed the research on pattern-based program repair in the literature. Nevertheless, we acknowledge that integrating more fix patterns may not necessarily lead to increased number of bugs that are correctly fixed. With too many fix patterns, the search space of fix patterns and patch candidates will explode. Eventually, the APR tool will produce a huge number of plausible patches, many of which might be validated before the correct ones [226]. A future research direction could be on the construction and curation of fix patterns database for APR.

Our strategy of fix pattern selection can be a threat to internal validity: it naïvely matches patterns based on the AST context around buggy locations. More advanced strategies would give a higher probability to select appropriate patterns to fix more bugs. Our approach to searching for donor code also carries some threats to validity: `TBar` focuses on the local buggy file, while previous works have shown that the adequate donor code, for some bugs, is available in other files [93, 226]. In future work, we will investigate the search of donor code beyond local files, while using heuristics to cope with the potential search space explosion. Finally, the selected benchmark for evaluation constitutes another threat to external validity for assessment. The performance achieved by `TBar` on Defects4J may not be reached on a bigger, more diverse and more representative dataset. To address this threat, new benchmarks such as Bugs.jar [198] and Bears [152] should be investigated.

7.5.2 Limitations

`TBar` selects fix patterns in a naïve way, it thus would be necessary to design a sophisticated strategy (such as bug symptom, bug type, or other information from bug reports) for fix pattern selection to reduce the noise from inappropriate fix patterns. Searching donor code for synthesis patches is another limitation of `TBar`, as the correct donor code for fixing some bugs is located in the code files that do not contain the bug [93, 226]. If `TBar` extends the donor code searching to other non-buggy code files, it will cause the search space explosion.

7.6 Related Work

Fault Localization. In general, most APR pipelines start with fault localization (FL), as shown in Figure 7.1. Once the buggy position is localized, ARP tools can mutate the buggy code entity to generate patches. To identify defect locations in a program, several automated FL techniques have been proposed [234]: slice-based [153, 231], spectrum-based [5, 190], statistics-based [134, 136], etc.

Spectrum-based FL is widely adopted in APR systems since they identify bug position at the statement level. It relies on the ranking metrics (e.g., Trantula [97], Ochiai [4]), Op2 [177], Barinel [5], and Dstar [233]) to calculate the suspiciousness of each statement. To debug Java program automatically, GZoltar [40] and Ochiai have been widely integrated into APR systems since their effectiveness has been demonstrated in several empirical studies [187, 208, 236, 241]. As reported by Liu et al. [140] and studied in this paper, this FL configuration still has a limitation on localizing bug positions. Therefore, researchers tried to enhance FL techniques with new techniques, such as predicate switching [239, 251] and test case purification [93, 242].

Patch Generation. Another key process of APR pipelines is searching for another shape of a program (i.e., a patch) in the space of all possible programs [126, 149]. If the search space is too small, it might not include the correct patches. [226]. To reduce this threat, a straightforward strategy is to expand the search space, however, which could lead to other two problems: (1) at worst, there still is no correct patch in it; and (2) the expanded search space includes more plausible patches that enlarge the possibility of generating plausible patches before correct ones [143, 226].

To improve repair performance, many APR systems have been explored to address the search space problem. Synthesis-based APR systems [148, 239, 240] explored to limit the search space on conditional bug fixes by synthesizing new conditional expressions with variables identified from the buggy code. Pattern-based APR tools [56, 91, 93, 104, 122, 124, 141, 144, 147, 199] are designed to purify the search space by following fix patterns to mutate buggy code entities with retrieved donor code. Other APR pipelines focus on specific search methods for donor code or patch synthesizing strategies, to address the search space problem, such as contract-based [43, 223], symbolic execution based [179], learning based [26, 75, 150, 196, 205, 227], and donor code searching [102, 164] APR tools. Various existing APR tools have achieved promising results on fixing real bugs, but there is still an opportunity to improve

the performance; for example, mining more fix patterns, improving pattern selection and donor code retrieving strategy, exploring a new strategy for patch generation, and prioritizing bug positions.

Patch Correctness. The ultimate goal of APR systems is to automatically generate a correct patch that can resolve the program defects. In the beginning, patch correctness is evaluated by passing all test cases [104, 124, 224]. However, these patches could be overfitting [125, 194] and even worse than the bug [202]. Since then, APR systems are evaluated with the precision of generating correct patches [93, 141, 226, 239]. Recently, researchers start to explore automated frameworks that can identify patch correctness for APR systems automatically [121, 238].

7.7 Summary

Fix patterns have been studied in various scenarios to understand bug fixes in the wild. They are further implemented in different APR pipelines to generate patches automatically. Although template-based APR tools have achieved promising results, no extensive investigation on the effectiveness fix patterns was conducted. We fill this gap in this work by revisiting the repair performance of fix patterns via a systematic study assessing the effectiveness of a variety of fix patterns summarized from the literature. In particular, we build a straightforward template-based APR tool, `TBar`, which we evaluate on the Defects4J benchmark. On the one hand, assuming a perfect fault localization, `TBar` fixes 74/101 bugs correctly/plausibly. On the other hand, in a normal/practical APR pipeline, `TBar` correctly fixes 43 bugs despite the noise of fault localization false positives. This constitutes a record performance in the literature on Java program repair. We expect `TBar` to be established as the new baseline APR system, leading researchers to propose better techniques for substantial improvement of the state-of-the-art.

8 Learning to Spot and Refactor Inconsistent Method Names

To ensure code readability and facilitate software maintenance, method names must be consistent with the corresponding method implementations. Debugging method names remains an important topic in the literature of detecting inconsistent method names and suggesting better ones. We propose a novel automated approach to debugging method names based on the analysis of consistency between method names and method code. The approach leverages deep feature representation techniques adapted to the nature of each artifact. Experimental results on over 2.1 million Java methods show that our proposed approach can effectively spot the inconsistent method names and suggest appropriate ones. Finally, we report on our success in fixing 66 inconsistent method names in a live study on projects in the wild.

This chapter is based on the work published in the following research paper:

- Kui Liu, Dongsun Kim, Tegawendé François D Assise Bissyande, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, pages 1–12. IEEE, 2019

Contents

8.1 Overview

"If you have a good name for a method, you don't need to look at the body." — Fowler et al. [62]

Names unlock the door to languages. In programming, names (i.e., identifiers) are pervasive in all program concepts, such as classes, methods, and variables. Descriptive names are the intuitive characteristic of objects being identified, thus, correct naming is essential for ensuring readability and maintainability of software programs. As highlighted by a number of industry experts, including McConnell [162], Beck [24], and Martin [155], naming is one of the key activities in programming.

Naming is a non-trivial task for program developers. Studies conducted by Johnson [95,96] concluded that identifier naming is the hardest task that programmers must complete. Indeed, developers often write poor (i.e., inconsistent) names in programs due to various reasons, such as lacking a good thesaurus, conflicting styles during collaboration among several developers, and improper code cloning [108].

Method names are the intuitive and vital information for developers to understand the behavior of programs or APIs [23, 53, 54, 67]. Therefore, inconsistent method names can make programs harder to understand and maintain [19, 20, 87, 119, 133, 212, 228], and may even lead to software defects [1, 2, 15, 36, 191]. Poor method names are indeed prone to be defective. For example, the commonly-used FindBugs [90] static analyzer even enumerates up to ten bug types related to method identifiers.

Figure 8.1 provides examples from project `AspectJ`[1] to illustrate how inconsistent names can be confusing about the executable behavior of a method. The name of the first method, `containsField`, suggests a question and is consistent with the method behavior which is about checking whether the `fieldsList` contains the target field `f`. The second method implements the search of a field in the target dataset and is thus consistently named `findField`. The third method is implemented similarly to the second method `findField`, but is named `containsField` as the first method. This name is inconsistent and can lead to misunderstanding of API usage.

```
1  public boolean containsField(Field f) {
2      return fieldsList.contains(f);
3  }
4
5  private ResolvedMember findField(ResolvedType resolvedType,String fieldName){
6      for(ResolvedMember field : resolvedType.getDeclaredFields()) {
7          if (field.getName().equals(fieldName)) {
8              return field;
9          }
10     }
11     return null;
12 }
13
14 public Field containsField(String name) {
15     for(Iterator e = this.field_vec.iterator(); e.hasNext();) {
16         Field f = (Field) e.next();
17         if (f.getName().equals(name)) {
18             return f;
19         }
20     }
21     return null;
22 }
```

Figure 8.1: Motivation examples taken from project `AspectJ`.

[1] `https://github.com/eclipse/org.aspectj`

As a preliminary study on the extent of the inconsistent method naming problem, we investigated posts by developers and users on fora and code repositories. We performed a search using composite conjunctions of "*method name*" and a category of keywords (i.e., *inconsistent, consistency, misleading, inappropriate, incorrect, confusing, wrong, bug* and *error*) to match relevant questions in StackOverflow[2] and commit logs in GitHub[3]. As a result, we managed to spot 5,644 questions and 183,901 commits, which are relevant to the issue of inconsistencies in method naming. Figures 8.2 and 8.3 show some excerpts of retrieved results.

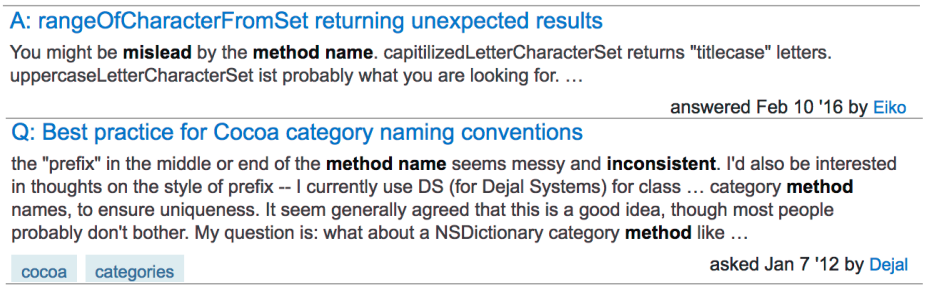


Figure 8.2: Excerpts of spotted questions about inconsistent method names in StackOverflow.

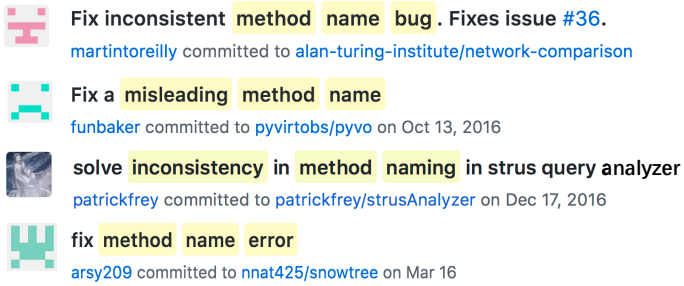


Figure 8.3: Excerpts of commit logs about inconsistent method names in GitHub.

Additionally, to assess the extent to which developers are prone to fix method names, we investigated the history of changes in all 430 projects collected for our experiments: in 53,731 commits, a method name is changed without any change to the corresponding body code. We further tracked future changes and noted that in 16% of the cases, the change is final (i.e., neither the method body nor the method name is changed again in later revisions of the project). These findings suggest that developers are indeed striving to choose appropriate method names, often to address consistency with the contexts of their code.

To debug method name, Høst and Østvold [89] explored method naming rules and semantic profiles of method implementations. Kim *et al.* [108] relied on a custom code dictionary to detect inconsistent names. Allamanis *et al.* introduced the NATURALIZE framework [9] learning the domain-specific naming convention from local contexts to improve the stylistic consistency of code identifiers with n-gram model [33]. Then, building on this framework, they proposed a log-bilinear neural probabilistic language model to suggest method and class names with similar contexts [10]. The researchers leveraged attentional neural networks [12] to extract local time-invariant and long-range topical attention features in a context-dependent way to suggest names for methods.

Overall, their context information is limited to local identifier sub-tokens and the data types of input and output. While the state-of-the-art has achieved promising results, a prime criterion of naming methods has not been considered: the implementation of methods that is a first-class feature to assess method naming consistency since method names should be mere summaries of methods' behavior [62]. Examples shown in Figure 8.1 illustrate the intuition behind our work:

[2]http://stackoverflow.com
[3]https://github.com

> *Methods implementing similar behavior in their body code are likely to be consistently named with similar names, and vice versa. It should be possible to suggest new names for a method, in replacement to its inconsistent name, by considering consistent names of similarly implemented methods.*

In this paper, we propose a novel automated approach to spotting and refactoring inconsistent method names. Our approach leverages Paragraph Vector [120] and Convolutional Neural Networks [161] to extract deep representations of method names and bodies, respectively. Then, given a method name, we compute two sets of similar names: the first one corresponds to those that can be identified by the trained model of method names; the second one, on the other hand, includes names of methods whose bodies are positively identified as similar to the body of the input method. If the two sets intersect to some extent (which is tuned by a threshold parameter), the method name is identified to be consistent, and inconsistent otherwise. We further leverage the second set of consistent names to suggest new names when the input method name is flagged as inconsistent.

To evaluate our proposed approach, we perform experiments with 2,116,413 methods of training data and 2,805 methods with changed names of test data, which are collected from 430 open source Java projects. Our experimental results show that the approach can achieve an F1-measure of 67.9% in the identification of inconsistent method names, representing an improvement of about 15 percentage points over the state-of-the-art. Furthermore, the approach achieves 34−50% accuracy on suggesting first sub-tokens and 16−25% accuracy on suggesting accurate full names for inconsistent method names, again outperforming the state-of-the-art. Finally, we report how our approach helped developers in fixing 66 inconsistent method names in 10 projects during a live study in the wild.

8.2 Background

This section briefly describes three techniques from the field of neural networks, namely Word2Vec [169], Paragraph Vector [120] and Convolutional Neural Networks [161]. Our approach relies on these techniques to achieve two objectives: (1) embedding tokens from method names and bodies into numerical vector forms, and (2) extracting feature representations for accurately identifying similar method names and bodies.

8.2.1 Paragraph Vector

Paragraph Vector is an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of texts, such as sentences [120]. This technique was proposed to overcome the limitations of bag-of-word [49] features which are known to (1) lose the order of words and (2) ignore the word semantics. Recent studies provide evidence that paragraph vector outperforms other state-of-the-art techniques [49, 214] for text representations [7, 120], and can effectively capture semantic similarities among words and sentences [50, 116, 117, 215, 229].

In our work, we use Paragraph Vector for training a model to compute similarities among method names (considering sequences of method name sub-tokens as sentences). We expect this model to take into account not only the lexical similarity but also the semantic similarity: for example, function names `containsObject` and `hasObject` should be classified as similar names since both of them describe the functionality of code implementation to check whether a set contains a specific object put in argument(s). We detail in later parts of this paper how method names are processed in our approach to feeding the Paragraph Vector algorithm.

8.2.2 Convolutional Neural Networks (CNNs)

CNNs are biologically-inspired variants of multi-layer artificial neural networks [161]. Initially developed and proven effectiveness in the area of image recognition, CNNs have gained popularity for handling various NLP tasks. For text classification, these deep learning models have achieved remarkable results [109, 221] by managing to capture the semantics of sentences for relevant similarity computation. Recently, a number of studies [13, 73, 74, 94, 176, 180, 189] have provided empirical evidence to support the *naturalness of software* [11, 86]. Thus, inspired by the *naturalness* hypothesis, we treat source code, in particular, method bodies, as documents written in natural language and to which we apply CNNs for code embedding purpose. The objective is to produce a model that will allow to accurately identify similar method code. A recent work by Bui et al. [34] has provided preliminary results showing that some variants of CNNs are even effective to capture code semantics so as to allow the accurate classification of code implementations across programming languages. In this study, we use LeNet5 [130], a specific implementation of CNNs, which consists of lower-layers and upper-layers (see Figure 8.5 for its architecture).

8.2.3 Word2Vec

When feeding tokens of a method body to CNNs, it is necessary to convert the tokens into numerical vectors. Otherwise, the size of a CNN's input layer would be too large if using one-hot encoding, or interpreting its output can be distorted if using numeric encoding (i.e., assigning a single integer value for each token). The machine learning community often uses vector representation for word tokens [45, 109, 120]. This offers two advantages: (1) a large number of (unique) tokens can be represented as a fixed-width vector form (dimensionality reduction) and (2) similar tokens can be located in a vector space so that the similar tokens can be dealt with CNNs in a similar way. Our approach uses Word2Vec [169] to embed tokens of method bodies.

Word2Vec[4] is a technique that encodes tokens into n-dimensional vectors [169, 170]. It is basically a two-layered neural network dedicated to process token sequences. The neural network takes a set of token sequences (i.e., sentences) as inputs and produces a map between a token and a numerical vector. The technique not only embeds tokens into numerical vectors but also places semantically similar words in adjacent locations in the vector space.

8.3 Approach

This section presents our approach to debugging inconsistent method names. As illustrated in Figure 8.4, it involves two phases: (1) training and (2) identification & suggestion. In the training phase, taking as input a large number of methods from real-world projects, it uses Paragraph Vector for method names and Word2Vec + CNNs for method bodies to embed them into numerical vectors (hereafter simply referred to as vectors), respectively. Eventually, two distinct vector spaces are produced and will be leveraged in the next phase. The objective of the training phase is thus to place similar method names and bodies into adjacent locations in each vector space.

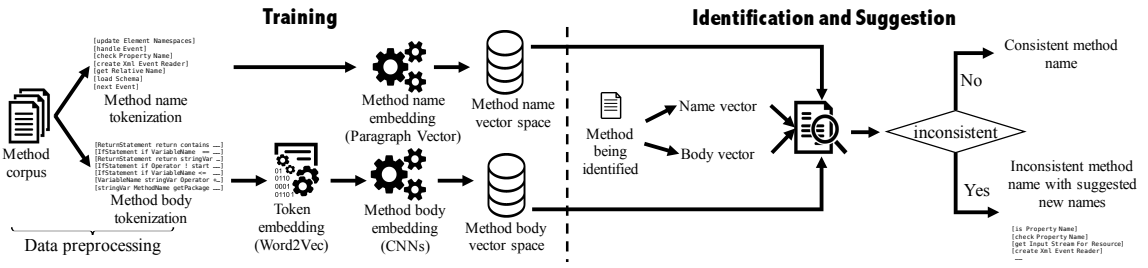


Figure 8.4: Overview of our approach to spotting and refactoring inconsistent method names.

[4] https://code.google.com/archive/p/word2vec/

The identification & suggestion phase determines whether a given method has a name that is consistent with its body by comparing the overlap between the set of method names that are close in the name vector space and the set of methods names whose bodies are close in the body vector space. When the overlap is $\emptyset$, the name is considered to be inconsistent with the body code and suggested with alternative consistent names.

Before explaining the details of these two phases, we first describe an essential step of data processing that adapts to the settings of code constructs.

8.3.1 Data Preprocessing

This step aims at preparing the raw data of a given method to be fed into the workflow of our approach. We consider the textual representations of code and transform them into tokens (i.e., basic data units) which are suitable for the deep representation learning techniques described in Section 8.2. Given that method names and bodies have different shapes (i.e., names are about natural language descriptions while bodies are focused on code implementations of algorithms), we propose to use tokenization techniques adapted to each:

- **Method name tokenization**: Method names are broken into sub-token sequences based on *camel case* and *underscore* naming conventions, and the obtained sub-tokens are brought to their lowercase form. This strategy has been proven effective in prior studies [9, 10, 12, 88, 89, 108]. For example, method names `findField` and `find_field` are tokenized into the same sequence [`find`, `field`], where `find` and `field` are respectively the first and second sub-tokens of the names.
- **Method body tokenization**: Method bodies are converted into textual token sequences by following the code parsing method proposed in our previous study [144]: this method consists in traversing the abstract syntax tree (AST) of a method body code with a depth-first search algorithm to collect two kinds of tokens: AST node types and raw code tokens. For example, the declaration statement "`int a;`" will be converted into a four-token sequence: [`PrimitiveType`, `int`, `Variable`, `a`]. Since noisy information of code (e.g., non-descriptive variable names such as `a`, `b`) can interfere with identifying similar code [131], all local variables are renamed as the concatenation of their data type with the string `Var`. Eventually, the previous declaration code will be represented by the sequence: [`PrimitiveType`, `int`, `Variable`, `intVar`].

8.3.2 Training

This phase takes tokens of method names and bodies in a code corpus to produce two numerical vector spaces that are leveraged to compute similarities, among method names, on the one hand, and among method bodies, on the other hand, for eventually identifying inconsistent names and suggesting appropriate names. Note that the objective is not to train a classifier whose output will be some classification label given a method name or body. Instead, we adopt the idea of unsupervised learning [81] and lazy learning [6] to embed method names and bodies.

Token sequences of method names are embedded into vectors by the paragraph vector technique described in Section 8.2.1 since token sequences of method names resemble sentences describing the methods. In contrast, all tokens in a method body are first embedded into vectors using Word2Vec. The embedded token vectors are then fed to CNNs to embed the whole method body into a vector, which will be used to represent each method body as a numerical vector.

8.3.2.1 Token Embedding for Method Bodies

As shown in Figure 8.4, tokens of method bodies are embedded into individual numerical vectors before they can be fed to the CNNs. To that end, the token embedding model is built as below:

$$\mathrm{TV}_B \leftarrow E_W(T_B) \tag{8.1}$$

where E_W is the token embedding function (i.e., Word2Vec [169] in our case) taking as input a training set of method body token sequences T_B. The output is then a token mapping function $\text{TV}_B : TW_B \rightarrow V_{BW}$, where TW_B is a vocabulary of method body tokens, and V_{BW} is the vector space embedding the tokens in TW_B.

After token embedding, a method body is eventually represented as a two-dimensional numerical vector. Suppose that a given method body b is represented by a sequence of tokens $T_b = (t_1, t_2, t_3, \ldots, t_k)$, where $t_i \in TW_B$, and V_b is a two-dimensional numerical vector corresponding to T_b. Then V_b is inferred as follows:

$$V_b \leftarrow l(T_b, \text{TV}_B) \tag{8.2}$$

where l is a function that transforms a token sequence of a method body into a two-dimensional numerical vector based on the mapping function TV_B. Thus, $V_b = (v_1, v_2, v_3, \ldots, v_k) \in V_B$, where $v_i \leftarrow \text{TV}_B(t_i)$ and V_B is a set of two-dimensional vectors.

Since token sequences of method bodies may have different lengths (i.e., k could be different for each method body), the corresponding vectors must be padded to comply with a fixed-width input layer in CNNs. Our approach follows the workaround tested by Wang *et al.* [222] and appends PAD vectors (i.e., zero vectors) to make all vector sizes consistent with the size of the longest one (see Section 8.4.2 for how to determine the longest one). For example, the left side of Figure 8.5 (See Section 8.3.2.2 for its description) shows how a method body is represented by a two-dimensional $n \times k$ numerical vector, where n is the vector size of each token and k is the size of the longest token sequence of bodies. Each row represents a vector of an embedded token, and the last two rows represent the appended zero vectors to make all two-dimensional vector sizes consistent.

8.3.2.2 Embedding Method Names and Bodies into Vectors

Vector spaces are built by embedding method names and bodies into corresponding numerical vectors. For method names, we feed the sub-token sequences (i.e., one sequence per method name) to a paragraph embedding technique. Specifically, we leverage the paragraph vector with distributed memory (PV-DM) technique [120], which embeds token sequences into a vector space as follows:

$$\text{NV}_{name} \leftarrow E_{PV}(T_N) \tag{8.3}$$

where E_{PV} is the paragraph vector embedding function (i.e., PV-DM), which takes as input a training set of method name sub-token sequences T_N. The output is a name mapping function $\text{NV}_{name} : T_N \rightarrow V_N$, where V_N is an embedded vector space for method names. This step is similar to classical word embedding with differences in the mapping relationships. The paragraph vector embeds a token sequence into a vector, while Word2Vec embeds a token into a vector.

For method bodies, we need another mapping function, where the input is a two-dimensional numerical vector for each method body. The output is a vector corresponding to a body. This mapping function is obtained by the formula below:

$$\text{VV}_{body} \leftarrow E_{BV}(V_B) \tag{8.4}$$

where E_{BV} is an embedding function (i.e., CNNs) that takes the two-dimensional vectors of method bodies (V_B) as training data and produces a mapping function (VV_{body}). Note that $V_B = \{V_{b_1}, V_{b_2}, V_{b_3}, \ldots, V_{b_m}\}$ is obtained by l (Equation 8.2), where V_{b_i} ($i \in [1, m]$) and m is the size of training data. VV_{body} is defined as $\text{VV}_{body} : V_B \rightarrow V'_B$, where V'_B is an embedded vector space of method bodies. Based on VV_{body}, we defined the body mapping function NV_{body} as:

$$\text{NV}_{body} : T_B \rightarrow V'_B \tag{8.5}$$

where NV_{body} is the composition of l and VV_{body} in Equations 8.2 and 8.4, respectively (i.e., $\text{NV}_{body} = (VV_{body} \circ l)(T_b) = VV_{body}(l(T_b))$). NV_{body} takes a token sequence of a method body and returns an embedded vector representing it.

Our approach uses CNNs [161] as the embedding function E_{BV} in Equation 8.5. Figure 8.5 shows the architecture of CNNs that our approach uses. The input is two-dimensional numeric vectors of method bodies as stated in Section 8.3.2.1. The two pairs of convolutional and subsampling layers are used to capture the local features of methods and decrease dimensions of input data. The network layers from the second subsampling layer to the subsequent layers are fully connected, which can combine all local features captured by convolutional and subsampling layers. We select the output of dense layer as the vector representations of method bodies, which synthesizes all local features captured by previous layers.

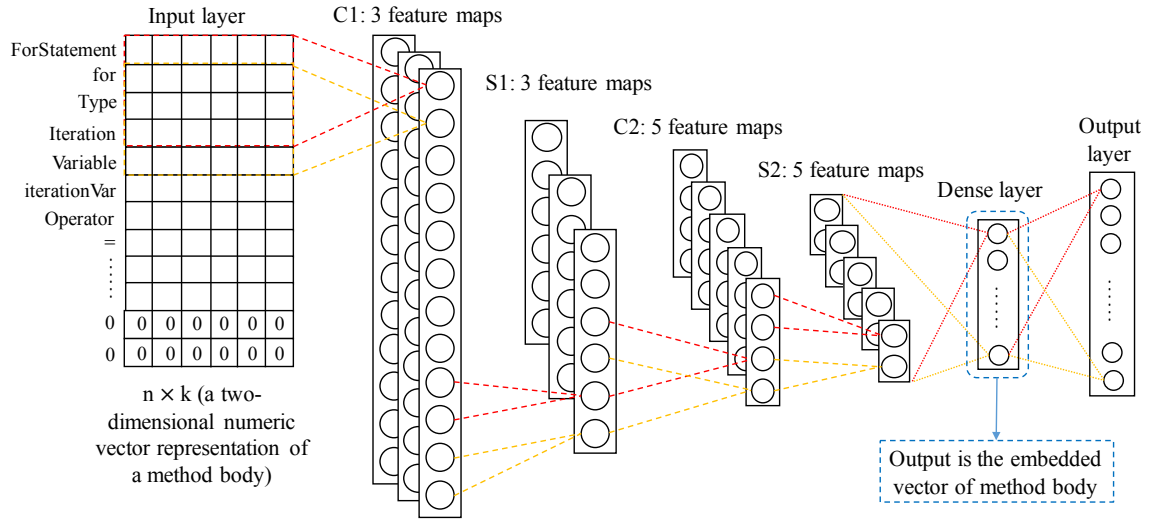


Figure 8.5: Architecture of CNNs [130] used in our approach to vectorize method bodies, where C1 and C2 are convolutional layers, and S1 and S2 are subsampling layers, respectively.

Note that vectors in the two vector spaces (i.e., V_N and V'_B) can be indexed by each method name. For a given body vector of a method, we can immediately find its corresponding name vector in the name vector space, and vice versa. This index facilitates the search of corresponding method names after locating similar method bodies for a given method.

8.3.3 Identification & Suggestion

This phase consists of two sub-steps. First, the approach takes a given method as a query of inconsistency identification. By leveraging the two vector spaces (i.e., V_N and V'_B) and the two embedding functions (i.e., NV_{name} and NV_{body}), it identifies whether the name of the given method is consistent with its body. Second, if the name turns out to be inconsistent, the approach suggests potentially consistent names for it from the names of similarly implemented methods.

8.3.3.1 Inconsistency Identification

For a given method m_i, we can take a set of adjacent vectors for its name (n_i) and body (b_i), respectively (i.e., $adj(n_i)$ and $adj(b_i)$). After retrieving the actual names (i.e., $name(*)$) corresponding to vectors in $adj(n_i)$ and $adj(b_i)$, we can compute the intersection between the two name sets as C_{full}:

$$C_{full} = name(adj(n_i)) \cap name(adj(b_i)) \tag{8.6}$$

If C_{full} is $\emptyset$, we consider b_i to be inconsistently named n_i. However, C_{full} in Equation 8.6 is too strict since it relies on exact matching. In other words, there should exist the same character sequences between two name sets. For example, suppose that there is `findField` in $name(adj(n_i))$ and `findElement` in $name(adj(b_i))$ with similar implementations. This relationship cannot be identified by C_{full} even if they have similar behavior of looking up something.

In the Java naming conventions[5], the first sub-token often indicates the key behavior of a method [38] (e.g., `get[...]()`, `contains[...]()`). Thus, if the key behavior of a given method is similar to those of other methods with similar bodies, we can regard that the name is consistent. Thus, we relax the condition of consistency. Instead of comparing the full name, we take the first sub-token of each name in the two name sets to get the intersection as below:

$$C_{relaxed} = first(name(adj(n_i))) \cap first(name(adj(b_i))) \tag{8.7}$$

where $first(*)$ is a function that obtains the first sub-token set by the same rule of method name tokenization described in Section 8.3.1. Other subsequent tokens are often project-specific. Therefore, those subsequent tokens would be different across projects even if their bodies are highly similar.

Algorithm 1 details the precise routine for checking whether the name n_i of a method is consistent with its body b_i or not. Our approach computes the cosine similarity for a given method to search for similar methods. After retrieving the embedded vectors of the name n_i and body b_i (cf. lines 3 and 4), the approach looks up the top k adjacent vectors in the respective vectors spaces for method names and bodies (cf. lines 6 and 10). Since threshold k can affect the performance of identification, our evaluation described in Section 8.5 includes an experiment where k values are varied.

Algorithm 1: Inconsistency identification and new names suggestion.

Input: target method (name and body): $m_i = (n_i, b_i)$
Input: threshold of adjacent vectors: k
Input: set of name vectors obtained from a training set: V_N
Input: set of body vectors obtained from a training set: V'_B
Input: indexes of actual names from all vectors $\forall V \in V_N$ or V'_B: $\text{Idx}_{name} : V \rightarrow N$
Input: function embedding a name to a vector: NV_{name}
Input: function embedding a body to a vector: NV_{body}
Output: pair of the consistency determinant of m_i (Boolean) and a set of suggested names: (c, SG_n), where SG_n is $\emptyset$ if c is $false$.

```
1  Function identify(m_i, V_N, V'_B)
2      // compute name and body vectors of m_i.
3      V_n := NV_name(n_i);
4      V'_b := NV_body(b_i);
5
6      // get adjacent name vectors similar to the name vector (V_n) of m_i.
7      NameV_adj := getTopAdjacent(V_n, V_N, k);
8      //get actual names for adjacent name vectors (NameV_adj).
9      Names^{n_i}_adj := NameV_adj.collect(Idx_name(∀V ∈ NameV_adj));
10
11     // get adjacent body vectors similar to the body vector (V'_b) of m_i.
12     BodyV_adj := getTopAdjacent(V'_b, V'_B, k);
13     // get actual names for adjacent body vectors (BodyV_adj).
14     Names^{b_i}_adj := BodyV_adj.collect(Idx_name(∀V ∈ BodyV_adj));
15
16     // take the first tokens of actual names for adjacent name and body vectors.
17     fT^{n_i}_adj := Names^{n_i}_adj.collect(tokenize_name(∀N ∈ Names^{n_i}_adj).first);
18     fT^{b_i}_adj := Names^{b_i}_adj.collect(tokenize_name(∀N ∈ Names^{b_i}_adj).first);
19
20     if fT^{n_i}_adj ∩ fT^{b_i}_adj is ∅ then
21         // m_i has an inconsistent name and suggest new names.
22         newNames := rankNames(Names^{b_i}_adj, BodyV_adj);
23         (c, SG_n) := (false, newNames);
24     else
25         // m_i has a consistent name.
26         (c, SG_n) := (true, ∅);
```

[5] `http://www.oracle.com/technetwork/java/codeconventions-135099.html`

After remapping the set of adjacent vectors to sets of the corresponding method names (cf. lines 8 and 12), the sets are processed to keep only first sub-tokens (cf. lines 14 and 15), since our approach uses $C_{relaxed}$ as specified in Equation 8.7 to compare the two sets of first tokens, $fT^{n_i}_{adj}$ and $fT^{b_i}_{adj}$ (cf. line 16). If their intersection is $\emptyset$, the approach suggests names for the given method body b_i (cf. Section 8.3.3.2 for details). Otherwise, our approach assumes that n_i is consistent with b_i.

8.3.3.2 Name Suggestion

Our approach suggests new names for a given method by providing a ranked list of the similar names (cf. line 18), with four ranking strategies as below:

- **R1**: This strategy purely relies on the similarities between method bodies. The names of similar method bodies ($\text{Names}^{b_i}_{adj}$) are ranked by the similarities to the given method body (between V'_b and BodyV_{adj}).
- **R2**: This strategy first groups the same names in $\text{Names}^{b_i}_{adj}$ since there might be duplicates. It then ranks distinct names based on the size of the associated groups. Ties are broken based on the similarities between method bodies as **R1**.
- **R3**: Similarly to **R2**, this strategy groups the same names in $\text{Names}^{b_i}_{adj}$. Then, the strategy computes the average similarity to b_i of each group and ranks the groups based on the average similarity, but the group sizes are not considered.
- **R4**: To avoid having highly ranked groups with a small size as per strategy **R3**, this strategy eventually re-ranks all groups produced in **R3** by downgrading all 1-size groups to the lowest position.

8.4 Experimental Setup

Empirical validation of the approach is performed through various experiments. Before describing the results and conclusions, we present the research questions and the data collection as well as details on the parameter settings in implementation to facilitate replication.

8.4.1 Research Questions

To evaluate the approach, we propose to investigate the following research questions (RQs):

- **RQ1**: *How effectively does the approach identify inconsistent method names?*
- **RQ2**: *Can the approach suggest accurate method names?*
- **RQ3**: *How does the approach compare with the state-of-the-art in terms of performance?*
- **RQ4**: *To what extent applying the approach in the wild produces debugging suggestions that are acceptable to developers?*

8.4.2 Data Collection

We collect both training and test data from open source projects from four different communities, namely Apache, Spring, Hibernate, and Google. We consider 430 Java projects with at least 100 commits, to ensure that these have been well-maintained.

Training data is constituted by all methods of these projects, after filtering out noisy data with criteria as below:

- *main* methods, constructor methods, and example methods[6] are ignored since they have the less adverse effect on program maintenance and understanding, and can pollute the results of searching for similar methods.

[6]The package, class or method name includes the keyword “example”, “sample” or “template”.

- Empty methods (i.e., abstract or zero-statement methods) have no implementation and thus are filtered out.
- Methods names without alphabetic letters (e.g., some methods are named "_") are removed as they are undescriptive.

As a result, 2,425,939 methods are collected.

In practice, we further limit the training data to methods with reasonable size, to avoid the explosion of code tokens which can degrade performance. The sizes of token sequences of collected method bodies range from 2 to 60,310. According to the sizes' distribution shown in Figure 8.6, most methods have less than 100 tokens. We focus on building the training data with methods containing at most 94 tokens, which is set based on the upper whisker value[7] from the boxplot distribution of method body token sequence sizes in Figure 8.6. The sizes beyond the upper whisker value are considered as outliers [63]. Eventually, 2,116,413 methods are selected to be the training data, as indicated in Table 8.1. Note that methods in the training data are not labeled as *consistent* or *inconsistent* since the objective of training is to construct a vector space of methods with *presumable*[8] consistent names.

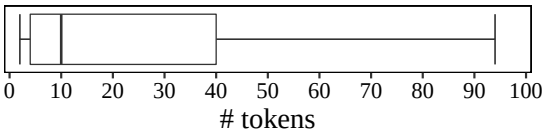


Figure 8.6: Sizes' distribution of collected method body token sequences.

Table 8.1: Size of datasets used in the experiments.

Classification	# Methods
All collected methods	2,425,939
Methods after filtering (for training)	2,116,413
Methods for testing	2,805

Test data is the oracle that we must constitute to assess the performance of our proposed approach. We build it by parsing the commit history of our subjects (i.e., 430 projects). Specifically, we consider:

- Methods whose names have been changed in a commit without any modification being performed on the body code;
- and the names and body code have become stable after the change (i.e., no more changes up to the current versions).

The first criterion allows to ensure that the change is really about fixing method names, and to retrieve their buggy and fixed versions. Overall, within commit changes, we identified 53,731 methods satisfying this criterion. The second criterion increases the confidence that the fixed version of the name is not itself found buggy later on. With this criterion, the number is reduced to 8,734 methods. We further observe that some method names are changed due to simple typos (cf. Figure 8.7). Such changes can constitute noise in the oracle. Given that our approach heavily relies on first sub-tokens of method names to hint at inconsistency, we conservatively ignore all change cases where this part is not changed. At this stage, the dataset still includes 4,445 buggy-fixed pairs of method names. The final selection follows the criterion used for collecting training data (i.e., no constructor or example methods, etc.). The final test data includes 2,805 distinct methods.

```
1 Commit 70106770ea61a5fe845653a0b793f4934cc00144
2 -     public double inverseCummulativeProbability(final double p) {
3 +     public double inverseCumulativeProbability(final double p) {
```

Figure 8.7: A typo fix for a method name in Apache commons-math.

[7]The upper whisker value is determined by 1.5 IQR (interquartile ranges) where IQR = 3rd Quartile − 1st Quartile, as defined in [63].
[8]The majority of the methods in the world have names that are likely to be consistent with their bodies.

To ensure that there is no data leakage [195] between training and test data that will artificially improve the performance of our approach, we eliminate from the training data all methods associated to the test data (i.e., there is no the same instance between 2,116,413 methods in training data and 2,805 methods in test data).

Our test data include method names for each of which we have two versions: the buggy name and the fixed one. To build our oracle, we need two sets, one for the *inconsistent* class and the other for the *consistent* class. We randomly divide our test data into two sets. In the first set, we consider only the buggy versions of the method names and label them as inconsistent. In the second set, we consider only the fixed versions and label them as consistent.

8.4.3 Implementation of Neural Network Models

The Paragraph Vector, Word2Vec, and CNNs models are implemented with the open source DL4J library[9], which is widely used across the research and practice in deep learning (800k+ people and 90k+ communities according to data on the Gitter networking platform[10]). These neural networks must be tuned with specific parameters. In this study, all parameters are set following the parameters setting proposed by Kim [109] and our previous work [144]. Their subjects and models are similar to ours, and their yielded models were shown to achieve promising results. Tables 8.2, 8.3, and 8.4 show the parameters used in our experiment for each model.

Table 8.2: Parameters setting of Paragraph Vector.

Parameters	Values
Min word frequency	1
Size of vector	300
Learning rate	0.025
Window size	2

Table 8.3: Parameters setting of Word2Vec.

Parameters	Values
Min word frequency	1
Size of vector	300
Learning rate	1e-2
Window size	4

Table 8.4: Parameters setting of CNNs.

Parameters	Values
# nodes in hidden layers	1000
learning rate	1e-2
activation (output layer)	softmax
pooling type	max pool
activation (other layers)	ReLU
optimization algorithm	stochastic gradient descent
loss function	mean absolute error

8.5 Evaluation

8.5.1 RQ1: Effectiveness of Inconsistency Identification

As the first objective of our approach is to identify *inconsistent* method names, we examine whether our approach effectively identifies methods with inconsistent names. We train the model with the collected training data and apply it to the separated test data whose collection was described in Section 8.4.2. Given that the performance of identification depends on the threshold value k representing the size of the sets of adjacent vectors (Lines 6 and 10 in Algorithm 1), we vary k as 1, 5, and $n \times 10$ with $n \in [1, 10]$.

[9] https://deeplearning4j.org/
[10] https://gitter.im/deeplearning4j/deeplearning4j

Inconsistent identification is a binary classification since the test data explained in Section 8.4.2 are labeled in two classes (IC: inconsistent, C: consistent). Thus, there are four possible outcomes: IC classified as IC (i.e, true positive=TP), IC classified as C (i.e., false negative=FN), C classified as C (i.e., true negative=TN), and C classified as IC (i.e., false positive=FP). We compute Precision, Recall, F1-measure and Accuracy for each class. The precision and recall for the class IC are defined as $\frac{|TP|}{|TP|+|FP|}$ and $\frac{|TP|}{|TP|+|FN|}$, respectively. Those for the class C are defined as $\frac{|TN|}{|TN|+|FN|}$ and $\frac{|TN|}{|TN|+|FP|}$, respectively. The F1-measure of each class is defined as $2 \cdot \frac{precision \cdot recall}{precision+recall}$, while the Accuracy is defined as $\frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|}$.

Table 8.5 provides the experimental results on the performance. Due to space limitation, we show the metrics for variations of k up to 40 (instead of 100). Overall, our approach yields an Accuracy metric ranging from 50.8% to 60.9% (for the presented results) and an F1-measure up to 67.9% for the inconsistent class. In particular, The approach achieves the highest performance when k=1 (i.e., the single most adjacent vector is considered). The general trend indeed is that the performance decreases as k is increased.

Table 8.5: Evaluation results of inconsistency identification.

	Evaluation metrics	k = 1	k = 5	k = 10	k = 20	k = 30	k = 40
Inconsistent	Precision (%)	**56.8**	53.7	53.3	53.3	49.9	49.7
	Recall (%)	**84.5**	55.9	46.7	46.7	28.8	33.6
	F1-measure (%)	**67.9**	54.8	49.7	49.7	36.5	40.1
Consistent	Precision (%)	**72.0**	55.9	54.2	54.2	51.4	51.4
	Recall (%)	38.2	53.7	60.7	60.7	**72.2**	67.4
	F1-measure (%)	49.9	54.8	57.3	57.3	**60.0**	58.3
Accuracy (%)		**60.9**	54.8	53.8	50.8	50.9	51.1

Since k determines the number of similar methods retrieving from the vector spaces of names and bodies in the training data, higher k value increases the probability of non-empty intersection (i.e., $\text{fT}^{n_i}_{adj} \cap \text{fT}^{b_i}_{adj}$, cf. line 16 in Algorithm 1). Thus, the recall of the *inconsistent* class tends to decrease as k is getting higher. In contrast, the recall of the *consistent* class increases for higher values of k.

Overall, the approach to identifying inconsistent method names can be tuned to meet the practitioners' requirements. When the criteria are to identify as many inconsistent names as possibles, k should be set to a low value.

8.5.2 RQ2: Accuracy in Method Names Suggestion

This experiment aims at evaluating the performance of our approach in suggesting new names for identified inconsistent names. The suggested names are ranked by a specifiable ranking strategy (either R1, R2, R3, or R4 as described in Section 8.3.3.2).

Prior studies compare the first tokens of suggested names [28, 89, 217] and oracles or token sets without considering token ordering [12, 14]. To ensure fair and comprehensive assessment, we consider three different scenarios to evaluate the performance of our approach. These scenarios are defined as follows:

- **T1** (Inconsistency avoidance): In this scenario, we evaluate to what extent the suggested names are different from the input buggy name n_i. The accuracy in this scenario is computed by $1 - \frac{\Sigma_{i \in ic} D\left(n_i^1, ft(\text{BodyV}_{adj})\right)}{|ic|}$, where n_i^1 is the first token of n_i. $ft()$ collects the first tokens of method names corresponding to each vector in BodyV_{adj} (similar body vectors shown at Line 10 in Algorithm 1). $D(*)$ checks whether the items in the second argument contain the first argument (if so, returns 1. Otherwise 0). ic is the set of inconsistent method names identified by our approach.

- **T2** (First-token accuracy): In this scenario, we evaluate to what extent the suggested names are identical to the name that developers proposed in debugging changes. We remind the reader that our test data indeed include pairs of buggy/fixed names extracted from code changes history (cf. Section 8.4.2). The accuracy is computed as $\frac{\Sigma_{i \in ic} D\left(rn(n_i)^1, ft(\text{BodyV}_{adj})\right)}{|ic|}$, where $rn(n_i)^1$ is the first token of the actual developer-fixed version of the name.
- **T3** (Full-name accuracy): In this scenario, we evaluate to what extent the full name of each suggested name is identical to the name that developers proposed in debugging changes. The accuracy in this scenario is computed as $\frac{\Sigma_{i \in ic} D(rn(n_i), fn(\text{BodyV}_{adj}))}{|ic|}$, where $rn(n_i)$ is the actual fixed version of the name and $fn(*)$ retrieves the full names of methods corresponding to each vector in BodyV_{adj}.

We perform different experiments, varying k. For these experiments, the approach produces new names for all methods identified as *inconsistent* (i.e., true_positive + false_positive). Thus, the results include the performance of false positives (i.e., names that are already consistent). We compute the performance based on the number of suggested names (varying the threshold value *thr*). For the ranking strategy R1, k is set only to 1 or 5 since higher values do not affect the results when $thr = 1$ or 5 (note that R1 produces the same number of suggested names with k). For other ranking strategies, k is set to 10, 20, 30, and 40 to have large numbers of suggested names that can be aggregated (as per ranking strategy working).

According to the results for **T1** listed in Table 8.6, our approach is highly likely to suggest names that are different from the identified inconsistent names with ranking strategies, even when k is high (>90% if *thr*=1 and >60% if *thr*=5). When matching the first tokens (cf. results for **T2**) and the full name (cf. results for **T3**), the ranking strategy R4 slightly outperforms others regardless of the k value, while its accuracy is ≈40%. *thr*=5 gives a better probability to find consistent names than *thr*=1. k=10 yields the best performance for ranking strategies R2 and R3 while ranking strategy R4 performs best when k=20 and *thr*=5 and k=40 and *thr*=1.

Table 8.6: Accuracy of suggesting method names with the four ranking strategies (i.e., R1, R2, R3 and R4).

Accuracy (%)		**k = thr**	**k = 10**			**k = 20**			**k = 30**			**k = 40**		
		R1	**R2**	**R3**	**R4**	**R2**	**R3**	**R4**	**R2**	**R3**	**R4**	**R2**	**R3**	**R4**
T1	thr=1	90.0	91.2	91.3	**76.1**	92.2	92.2	75.9	93.0	92.9	75.8	**93.7**	**93.6**	75.9
	thr=5	69.0	**66.4**	**66.4**	**66.1**	64.1	64.0	61.8	64.9	64.9	61.3	66.1	66.1	61.5
T2	thr=1	23.4	**23.2**	**23.0**	24.1	21.5	21.5	24.1	19.3	19.3	24.0	17.2	17.2	**24.2**
	thr=5	35.7	**39.4**	**39.4**	39.7	38.5	38.6	**40.8**	37.3	37.2	40.6	36.5	36.3	40.1
T3	thr=1	10.7	**11.0**	**10.9**	10.9	10.9	10.9	11.1	10.6	10.5	11.3	10.2	10.1	**11.5**
	thr=5	17.0	**18.7**	**19.0**	19.2	17.7	17.8	**19.5**	16.9	16.9	19.4	16.6	16.6	19.2

The best case of each ranking strategy in each row is highlighted as **bold**.

The results above show that: (1) ranking strategies R2, R3, and R4 perform better than R1 but they need more candidates, (2) higher k values do not increase the accuracy of name suggestion, and (3) more number of suggested names (i.e., higher *thr* values) would improve the accuracy but users of our approach will need to look up more names.

Note that it is promising that achieving ≈20% and ≈40% accuracy (for first-token / T2) when looking up only top-1 and top-5 suggestions, respectively. Suggesting exact first tokens of method names is challenging since there are a large number of available words for the first tokens of method names. Finding the exact full name of a method is even more challenging since full method names are often very project-specific [86]. Our approach achieves ≈10% and ≈20% accuracy, respectively for *thr*=1 and *thr*=5.

8.5.3 RQ3: Comparison Against the State-of-the-art Techniques

We compare our approach with two state-of-the-art approaches in the literature which are based on the n-gram model [210] and the convolutional attention network (CAN) model [12]. The latter includes two sub-models: *conv_attention*, which uses only the pre-trained vocabulary, and *copy_attention*, which can copy tokens of input vectors (i.e., tokens in a method body). These techniques are selected since they are the most recent approaches for method name debugging. Given that the n-gram model approach by Suzuki et al. [210] cannot suggest full names or even the first token of methods, we compare against them with respect to the performance of inconsistent name identification. The CAN model, on the other hand, does not explicitly identify name inconsistency. Instead, the model suggests names for any given method. Thus, in this experiment, we make the CAN model and our approach suggest names for all test data (2,805 buggy method names). For both techniques, we use the same training data described in Section 8.4.2. It should be noted that while the tool for the CAN model has been made available by the authors, we had to replicate the n-gram models approach, in a best effort way following the details available in [210].

Table 8.7 shows the comparison results with the n-gram model [210]. While the performance of the n-gram model stays in a range from 51.5-54.2% for all measures, our approach outperforms the model when k=1 and 5. With k=1, the im-provement is up to 33 percentage points. In particular, our app-roach achieves a higher F1-measure by 15 percentage points.

Table 8.7: Comparison results of identifying inconsistent method names against the n-gram model [210].

Evaluation Metrics	Our Approach			n-gram Model
	k = 1	k = 5	k = 10	
Precision	**56.8%**	53.7%	53.3%	53.3%
Recall	**84.5%**	55.9%	46.7%	51.5%
F1-measure	**67.9%**	54.8%	49.7%	52.4%
Accuracy	**60.9%**	54.8%	53.8%	54.2%

To compare our approach against the CAN model [12], we propose two evaluations. The first follows the evaluation strategy proposed by the authors themselves in their paper. The second evaluation is based on our own strategies already explored for RQ2 (cf. Section 8.5.2).

Table 8.8 shows the performance based on the *per-sub-token basis* metric, which is the evaluation metric used by the authors originally to present the performance of the CAN model [12]. This metric estimates to what extent sub-tokens of method names can be correctly suggested without considering their order within the method names. We compute precision, recall, and F1-measure of correctly suggesting sub-tokens.

Table 8.8: Comparison of the CAN model [12] and our approach based on the per-sub-token criterion [12].

	Precision		Recall		F1-meansure	
	thr = 1	thr = 5	thr = 1	thr = 5	thr = 1	thr = 5
conv_attention	23.2%	36.5%	8.1%	13.1%	11.7%	18.7%
copy_attention	28.4%	**67.0%**	10.0%	27.5%	14.4%	37.9%
R1 (k = thr)	29.7%	38.6%	27.4%	36.7%	28.5%	37.6%
R2 (k = 10)	30.1%	39.6%	**27.6%**	37.2%	**28.8%**	38.3%
R3 (k = 10)	**30.2%**	39.9%	**27.6%**	**37.6%**	**28.8%**	**38.7%**
R4 (k = 10)	27.2%	38.6%	25.2%	**37.6%**	26.2%	38.1%

When applying the per-sub-token basis, our approach outperforms the CAN model in all configurations, except for the precision of *copy_attention* with *thr*=5 (cf. Section 8.5.2). While the precision of our approach can be higher by up to 7 percentage points, we achieve substantial performance improvement in terms of recall and F1-measure, with up to 15 percentage points margin.

Table 8.9 presents the comparison results when applying the three evaluation strategies of RQ2, described in Section 8.5.2. Regardless of the evaluation strategy, our approach outperforms the CAN models. Notably, our approach achieves **16∼25%** accuracy for **T3** (i.e., full name suggestion) while the CAN model is only successful for at most **1.1%**. Note that specific values of accuracy in Table 8.9 are different from Table 8.6 since, in the experiment for RQ3, our approach suggests names for all test data.

Table 8.9: Comparison of the CAN model [12] and our approach based on three evaluation scenarios.

Accuracy	**T1**		**T2**		**T3**	
	thr = 1	thr = 5	thr = 1	thr = 5	thr = 1	thr = 5
conv_attention	78.4%	27.6%	22.3%	33.6%	0.3%	0.6%
copy_attention	77.2%	38.9%	23.5%	44.7%	0.4%	1.1%
R1 (k = thr)	86.9%	**69.7%**	36.4%	47.2%	16.5%	22.9%
R2 (k = 10)	88.5%	67.5%	34.8%	50.2%	**17.0%**	25.4%
R3 (k = 10)	**88.6%**	67.5%	34.7%	50.3%	16.9%	25.5%
R4 (k = 10)	77.0%	67.3%	**35.4%**	**50.5%**	16.0%	**25.7%**

While state-of-the-art techniques directly train a classifier for identification or a neural network for suggestion by using a set of training data, our approach first transforms method names and bodies into vectors by using neural networks and then searches for similar vectors by computing distances between them. In that sense, our approach is implemented based on unsupervised learning. Overall, the results imply that looking up similar methods in vector spaces is more effective both for identification and suggestion than other techniques.

8.5.4 RQ4: Live Study

To investigate practicability of our approach to debugging inconsistent method names (**RQ4**), we conduct a live study on active software projects: we submit pull requests of renaming suggestions from our approach, and assess acceptance rates. For this experiment, we randomly sample 10% of the training data to be used as test data. Indeed the labeled test data collected for previous experiments represent cases where developers debugged the method names. The remaining 90% of method names now constitute the training data for this phase. We apply this version of our approach to the target subjects to identify whether they have inconsistent names (using k=20 as per result of previous experiments). Overall, 4,430 methods among the 211,642 methods in the test set have been identified as inconsistent by our approach. Given that we cannot afford to spam project maintainers with thousands of pull requests, we randomly select 100 cases of identified inconsistent methods. We then collect the ranked list of suggested names for each of the 100 methods: we use thr=5 with ranking strategy R4 since these parameters show the best performance for full name suggestion (T3). From each ranked list of suggested names, we select the top-1 name and prepare a patch that we submit as a pull request to the relevant project repository.

Table 8.10: Results of live study.

Agree			**Agree but not fixed**		**Disagree**	**Ignored**	**Total**
Merged	Approved	Improved	Cannot	Won't			
40	26	4	1	2	9	18	100

As indicated in Table 8.10, developers agreed to merge the pull requests for renaming 40 out of the 100 methods. 26 renaming suggestions have been validated and approved (based on developers' reply) by developers, but the pull requests have not been merged (as of submission date) since some projects systematically apply unit test and complete review tasks of external changes before accepting them into the main branch. Four inconsistent method names have also been fixed after improving our suggested names. Interestingly, one developer used our suggestion as a renaming pattern to fix six (6) similar cases other than the ones submitted in our pull requests. Furthermore, some developers have

welcomed our suggestions on inconsistent method names and showed interest in applying even more suggestions from our approach, given that it seems to provide more meaningful names than their current names.

We also report on cases where developers did not apply our suggested name changes. In 1 case, the developers *could not* merge the pull request as it would break the program: the method is actually an overridden method from another project. The developer nevertheless agreed that our suggested name was more intuitive. For 2 methods, developers agree that the suggested names are appropriate but they *would not* make the changes as the names are not in line with inner-project naming conventions. For nine methods, however, the pull requests are rejected since developers judge the original method names to be more meaningful than the suggested ones. The remaining 18 cases are simply *ignored*: we did not receive any reply up to the date of submission. We summarize developers' feedback as follows:

1. Some method names should follow the naming convention of specific projects. This is a threat to the validity of our study since it is implemented in a cross-project context.
2. Some method should be named considering the class names. E.g., in a class named "XXXBuilder", the developers do not want to name a method as "build", although the method builds a new "XXXBuilder" object.

8.6 Discussion

8.6.1 Naming based on Syntactic and Semantic Information

As stated in Section 8.1, our approach is based on the assumption that similar method implementations might be associated with similar method names. However, there could be several different definitions of similarity. While our approach relies on the syntactic similarity of method bodies (although, using AST tokens), one can use dynamic information (e.g., execution traces) to compare different method implementations as experimented for the detection of semantic (i.e., type-4) code clones [105, 209]. However, obtaining dynamic information is not scalable. Test cases are not always available and running concolic execution is still expensive. Although we can leverage code-to-code search techniques [106], their precision is not sufficient for inconsistent name detection. Thus, we leverage only static and syntactic information in our approach and rely on deep learning representations that have been shown to be effective capturing semantics even for code [14, 34].

8.6.2 Threats to Validity

A threat to external validity is in the training data since it is impossible to absolutely ensure that all methods in training data have consistent names. To address this threat, we collect training data from the well-maintained open source projects with high reputation. Although the number of projects may not be representative of the whole universe, it is the largest dataset used in published literature about debugging method names. Our live study further demonstrates that the training set is sufficient to build a good model. Other threat to external validity is the typos and abbreviations in method names that can noise method name embedding and suggestion.

Threats to internal validity include the limitation of parsing method names since some method names are named without following camel case or underscore naming convention. It is challenging to parse this kind of method names. This threat could be reduced by developing more advanced method name parse tools with natural language processing. Another threat to validity is the size of data set for testing since the test data is no less than 10% for evaluation in recent machine learning and natural language process literature, where the training and test data are split from collected data. In our study, test data must be actual fixed method names to evaluate the performance of debugging

inconsistent method names, but projects used for training do not have such a high number of fixed method names to satisfy the requirement of the balanced training and test data. Manually mutating method names could enlarge test data, but it could bias the assessing results. Thus, collecting more actual fixed method names from other projects are included in our future work.

8.7 Related Work

There have been several empirical studies [35, 37–39] investigating the impact of a naming scheme on program comprehension, readability, and maintainability. Takang *et al.* [212] and Lawrie *et al.* [119] conducted empirical studies on code identifiers and concluded that inconsistent names can make code harder to understand and maintain. Caprile and Tonella [41] analyzed function identifiers from the lexical, syntactical and semantic structure, and reported that identifiers can be decomposed into fragments and further classified into several lexical categories. Liblit *et al.* [133] examined how human cognition is reflected in naming things of programs.

Several approaches have been presented to detect inconsistent identifiers. Deissenboeck and Pizka [53], and Lawrie *et al.* [118] relied on the manual mapping between names and domain concepts to detect inconsistent identifiers in code. Binkley *et al.* [28] developed a tool with part-of-speech tagging to identify field identifiers that violate accepted patterns.

Even after automatically detecting inconsistent names, developers may have difficulties in debugging or refactoring inconsistent names. The ultimate goal of debugging names is to automatically replace inconsistent names into consistent ones rather than just helping identifier naming. Haiduc *et al.* [78] used natural language summarization techniques and the lexical and structural context in code to improve code comprehension [77]. Sridhara *et al.* [207] designed an automatic technique for summarizing code with the idioms and structure in a method. Lucia *et al.* [52] proposed an IR-based approach to improve program comprehension with the textual similarity between the code under development and related artefacts. Høst and Østvold [89] used method naming rules and semantic profiles of method implementations to debug method names.

Recently, Allamanis *et al.* [10, 12] leveraged deep learning techniques to suggest method names with local contexts, which are similar to this paper on embedding method names and bodies. Their work learns method body features from code sub-tokens, this paper further consider code nodes at abstract syntax tree level since they can capture code semantic information [176]. This paper just compared against the work [12] since both of them are from the same group and the work [12] presents two more advanced models. This paper performed various evaluations on the actual fixed method names which were not done in [10, 12], with a large sample of 430 projects against the 20/10 projects in their work. In addition, this paper tried various configurations and strategies and used various indicators for method name suggestions, which was not exactly the same as they did. Thus, our conclusions are likely to be more solid than those in their work. Furthermore, we performed a live study (not done in their work) and showed the technique has strong potential to be useful by actually fixing 66 inconsistent method names in the wild.

8.8 Summary

Method names are key to readable and maintainable code, but it is not an easy task to give an appropriate name to a method. Thus, many methods have inconsistent names, which can impede the readability and maintainability of programs and even lead to some defects. To reduce the manual efforts of resolving inconsistent method names, we propose a novel approach to debugging inconsistent method names by leveraging similar methods with deep learning techniques. Our experimental results show that the performance of our approach achieves an F1-measure of 67.9% on identifying inconsistent method names, improving about 15 percentage points over the state-of-the-art. On

suggesting appropriate first sub-tokens and full names for inconsistent method names, it achieves 34−50% and 16−25% accuracy respectively, outperforming the state-of-the-art as well. We further report that our approach helps developers to fix 66 inconsistent method names in the wild. The tool and data of in our study are available at `https://github.com/SerVal-DTF/debug-method-name`.

9 Conclusions and Future Work

In this chapter, we revisit the main contributions of this dissertation and present potential future research directions.

Contents

9.1 Conclusions

In this work, we presented techniques for boosting automated program repair targeting at Java programs by two scenarios: program bugs and method naming bugs. More specifically, we scheduled this dissertation in three parts: 1) Closer looking at real world patches; 2) automated program repair with fix patterns; and 3) spotting and refactoring inconsistent method names. We now detail them as below.

Regarding the first part, our objective is to provide a fine-grained view on real world patches, to further deepen the knowledge on patch generation for APR. Through a systematic study on 16,450 bug fix commits from seven Java open-source projects, we find that there are opportunities for APR techniques to target at code elements that have not yet been investigated. We also find that a small number of statement and expression types are recurrently impacted by real-world patches, and expression level granularity could reduce the search space of finding fix ingredients for similar bugs. We further discuss nine insights into tuning APR tools, challenges and possible resolves through investigating research questions around the actual locations of buggy code and repair actions at the AST level. Additionally, modifications of method names are considered as bugs fixed by some developers.

For the second part, we presented four studies to support automated program repair, where all of them have leveraged fix patterns for patch generation. In particular, we first investigated the fault localization impact on benchmarking automated program repair systems, which provides a clear view in the fault localization settings of the state-of-the-art APR systems and explores the performance variations that can be achieved by "tweaking" the fault localization step. On the other hand, we proposed a deep learning based fix pattern mining for static analysis bugs. Our evaluation shows that our proposed method can effectively mine fix patterns for static analysis bugs and the mined fix patterns could be used to fix similar bugs. Based on these mined fix patterns, we further proposed an automated program repair system, Avatar, to investigate the possibility in an APR scenario for semantic bugs of leveraging fix patterns that address violations by static bug detection tools. Avatar can achieve promising repair performance that is comparable to the state-of-the-art approaches in the literature. Similarly, we implemented a fix template based automated program repair system, TBar, to revisit the performance of template-based APR to build comprehensive knowledge about the effectiveness of fix patterns, and to highlight the importance of complementary steps such as fault localization or donor code retrieval. We thoroughly evaluate TBar on the Defects4J benchmark. In particular, we assess the actual qualitative and quantitative diversity of fix patterns, as well as their effectiveness in yielding plausible or correct patches.

As a final part, we presented a deep learning based approach to debugging inconsistent method names. To ensure code readability and facilitate software maintenance, program methods must be named properly. In particular, method names must be consistent with the corresponding method implementations. Debugging method names aims at detecting inconsistent method names and suggesting better ones. Our proposed approach leverages deep feature representation techniques adapted to the nature of each artifact to spot and refactor inconsistent method names in programs. Experimental and live study results show that our proposed approach can effectively debug inconsistent method names in the real-world code bases.

9.2 Future Work

We now summarize potential future directions that are in line with this dissertation.

1. **Patch prioritization in APR systems.** Heuristic-based repair (including *fix pattern based repair*) will construct a search space for patch generations. In theory, the state-of-the-art heuristic based APR systems can generate correct patches for some bugs, but they failed to

correctly fix these bugs. It is limited by two reasons: 1) fault localization impact: the plausible but incorrect patches are generated for the suspicious but non-buggy positions before the exact bug position; and 2) APR-intrinsic limitation: the plausible but incorrect patches are generated before the correct one for the exact bug position. Therefore, improve the patch prioritization for APR systems would be a promising future work to boost APR.

2. **Donor code searching for patch generation.** When fixing bugs with fix patterns, it needs to search for the appropriate donor code for generating patch candidates. In this dissertation, the donor code search is only explored in the class file where the bug is located, but it is insufficient for fixing the bugs whose donor code is located in the non-buggy class files. Thus, in future work, it is necessary to explore an effective donor code searching strategy to improve the repair performance of fix pattern based APR systems.
3. **Flexible fix pattern based APR.** Fix pattern based APR systems have shown to be an effective way of successfully addressing real bugs. However, fix patterns of APR systems in the literature are manually implemented by practitioners with pre-defined rules. Practitioners need to do it again when new fix patterns are available. It would be another promising future work to explore fix pattern based APR systems that are flexible to any new fix patterns without human interference.
4. **Reliable APR system for real bugs.** In the APR community, APR systems are commonly evaluated on a benchmark that consists of real bugs collected by practitioners. However, the bugs in such benchmark are manipulated by adding bug-triggering test cases. Unfortunately, the readily available bug-triggering test cases still do not hold in practice. Therefore, such real bugs are in dire need of reliable APR systems which can address them without adding bug-trigger test cases.

List of papers, tools & services

Papers included in this dissertation:

- Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. A closer look at real-world patches. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, pages 275–286. IEEE, 2018
- Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018
- Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, pages 102–113. IEEE, 2019
- Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, pages 1–12. IEEE, 2019
- Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42. ACM, 2019
- Kui Liu, Dongsun Kim, Tegawendé François D Assise Bissyande, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, pages 1–12. IEEE, 2019

Papers not included in this dissertation:

- Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*, pages 658–662. IEEE, 2018
- Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2019
- Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: bug report driven program repair. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325. ACM, 2019
- Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 2019
- Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. D&c: A divide-and-conquer approach to ir-based bug localization. *arXiv*, 2019

Tools and Datasets:

- **AVATAR** - `https://github.com/SerVal-DTF/AVATAR`
- **Debug-method-name** - `https://github.com/SerVal-DTF/debug-method-name`
- **fixminer-APR** - `https://github.com/SerVal-DTF/fixminer-APR`
- **FixPatternMiner** - `https://github.com/FixPattern/findbugs-violations`
- **kPAR** - `https://github.com/SerVal-DTF/FL-VS-APR/tree/master/kPAR`
- **iFixR** - `https://github.com/SerVal-DTF/iFixR`
- **JavaCodeParser** - `https://github.com/AutoProRepair/JavaCodeParser`
- **LSRepair** - `https://github.com/AutoProRepair/LSRepair`
- **PatchParser** - `https://github.com/AutoProRepair/PatchParser`
- **TBar** - `https://github.com/SerVal-DTF/TBar`

Services (External Reviewer):

- SANER'19, QRS'19, ICPC'19, ISSTA'19, ICSME'19, ASE'2019.
- ASWEC'18, ASE'18, ASE-DEMO'18, QRS'18, Codespy'18, ISSTA'18, ICSME'18, ICSE'18.
- ICST'17, ICPC'17, SANER'17.
- Information and Software Technology
- World Wide Web Journal

Bibliography

[1] Surafel Lemma Abebe, Venera Arnaoudova, Paolo Tonella, Giuliano Antoniol, and Yann-Gael Gueheneuc. Can lexicon bad smells improve fault prediction? In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 235–244. IEEE, 2012.

[2] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. The effect of lexicon bad smells on concept location in source code. In *Proceedings of the 11th International Working Conference on Source Code Analysis and Manipulation*, pages 125–134. IEEE, 2011.

[3] Rui Abreu, Arjan JC Van Gemund, and Peter Zoeteweij. On the accuracy of spectrum-based fault localization. In *Proceedings of the 2007 Testing: Academic and Industrial Conference Practice and Research Techniques: MUTATION*, pages 89–98. IEEE, 2007.

[4] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[5] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE Computer Society, 2009.

[6] David W. Aha. *Lazy learning*. Springer, Washington, DC, 1997.

[7] Qingyao Ai, Liu Yang, Jiafeng Guo, and W Bruce Croft. Analysis of the paragraph vector model for information retrieval. In *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*, pages 133–142. ACM, 2016.

[8] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. What's a typical commit? a characterization of open source software repositories. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 182–191. IEEE, 2008.

[9] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

[10] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

[11] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4):81, 2018.

[12] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the International Conference on Machine Learning*, pages 2091–2100. JMLR.org, 2016.

[13] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 2123–2132. JMLR.org, 2015.

[14] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. In *Proceedings of the 46th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 3, pages 40:1–40:29. ACM, 2019.

[15] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. A systematic evaluation of api-misuse detectors. *arXiv preprint arXiv:1712.00242*, 2017.

[16] John Anvik, Lyndon Hiew, and Gail C Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[17] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[18] Apache. Maven. `https://maven.apache.org/`, Last Accessed: Nov. 2017.

[19] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016.

[20] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gael Gueheneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014.

[21] Pavel Avgustinov, Arthur I Baars, Anders S Henriksen, Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. Tracking static analysis violations over time to capture developer characteristics. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 437–447. ACM, 2015.

[22] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317. ACM, 2014.

[23] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, 2014.

[24] Kent Beck. *Implementation patterns*. Pearson Education, 2007.

[25] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[26] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*, pages 60–70. ACM, 2018.

[27] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.

[28] Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 203–206. ACM, 2011.

[29] Tegawendé F Bissyandé. Harvesting fix hints in the history of bugs. *arXiv preprint arXiv:1507.05742*, 2015.

[30] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 117–128. ACM, 2017.

[31] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Technique Report in University of Cambridge, UK*, 2013.

[32] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522. ACM, 2017.

[33] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, 1992.

[34] Nghi DQ Bui, Lingixao Jiang, and Yijun Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. *arXiv preprint arXiv:1710.06159*, 2017.

[35] Simon Butler. Mining java class identifier naming conventions. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1641–1643. IEEE, 2012.

[36] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of 16th Working Conference on Reverse Engineering, WCRE'09.*, pages 31–35. IEEE, 2009.

[37] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

[38] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Mining java class naming conventions. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 93–102. IEEE, 2011.

[39] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. INVocD: identifier name vocabulary dataset. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 405–408. IEEE, 2013.

[40] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381. ACM, 2012.

[41] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 112–122. IEEE, 1999.

[42] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. Automatic recovery from runtime failures. In *Proceedings of the 35th International Conference on Software Engineering*, pages 782–791. IEEE, 2013.

[43] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 637–647. IEEE, 2017.

[44] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the 32nd International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[45] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. pages 1724–1734, 2014.

[46] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*, 2012.

[47] Zack Coker and Munawar Hafiz. Program transformations to fix c integers. In *Proceedings of the 35th International Conference on Software Engineering*, pages 792–801. IEEE, 2013.

[48] Tom Copeland. Pmd Applied, 2005.

[49] George E Dahl, Ryan P Adams, and Hugo Larochelle. Training restricted boltzmann machines on word observations. *arXiv preprint arXiv:1202.5695*, 2012.

[50] Andrew M Dai, Christopher Olah, and Quoc V Le. Document embedding with paragraph vectors. *arXiv preprint arXiv:1507.07998*, 2015.

[51] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554. ACM, 2009.

[52] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering*, 37(2):205–227, 2011.

[53] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[54] Florian Deissenboeck and Markus Pizka. Concise and consistent naming: ten years later. In *Proceedings of the 23rd International Conference on Program Comprehension*, pages 3–3. IEEE, 2015.

[55] Kinga Dobolyi and Westley Weimer. Changing java's semantics for handling null pointer exceptions. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 47–56. IEEE, 2008.

[56] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*, pages 349–358. IEEE, 2017.

[57] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the IEEE/ACM 11th International Workshop in Automation of Software Test*, pages 85–91. IEEE, 2016.

[58] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 8th ACM symposium on Operating Systems Principles*, volume 35, pages 57–72. ACM, 2001.

[59] Jon Eyolfson, Lin Tan, and Patrick Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, 2014.

[60] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[61] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 450–457. IEEE, 2011.

[62] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[63] Michael Frigge, David C Hoaglin, and Boris Iglewicz. Some implementations of the boxplot. *The American Statistician*, 43(1):50–54, 1989.

[64] Zachary P Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[65] Gousios Georgios and Spinellis Diomidis. Ghtorrent. `http://ghtorrent.org/halloffame.html`, Last Accessed: Nov. 2017.

[66] Daniel M German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[67] Malcom Gethers, Trevor Savage, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. CodeTopics: which topic am i coding now? In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1034–1036. ACM, 2011.

[68] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 19–30. ACM, 2019.

[69] Github. Pmd: an extensible cross-language static code analyzer. `https://pmd.github.io/`, Last Accessed: Nov. 2017.

[70] Github. semmle. `https://semmle.com/`, Last Accessed: Nov. 2017.

[71] Yoav Goldberg and Omer Levy. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[72] Inexpensive Program Analysis Group. Splint. `http://splint.org/`, Last Accessed: Nov. 2017.

[73] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944. ACM, 2018.

[74] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[75] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 1345–1351. AAAI Press, 2017.

[76] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 317–328. ACM, 2018.

[77] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226. ACM, 2010.

[78] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 35–44. IEEE, 2010.

[79] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 152–161. ACM, 2014.

[80] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[81] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Unsupervised learning. In *The Elements of Statistical Learning*, pages 485–585. Springer, 2009.

[82] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 41–50. ACM, 2008.

[83] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *Proceedings of the 2nd International Conference on Software Testing Verification and Validation*, pages 161–170. IEEE, 2009.

[84] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011.

[85] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 121–130. IEEE, 2013.

[86] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 837–847. IEEE, 2012.

[87] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. Shorter identifier names take longer to comprehend. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*, pages 217–227. IEEE, 2017.

[88] Einar W Høst and Bjarte M Østvold. The java programmer's phrase book. In *Proceedings of the First International Conference on Software Language Engineering*, pages 322–341. Springer, 2008.

[89] Einar W Høst and Bjarte M Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 294–317. Springer, 2009.

[90] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[91] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 12–23. ACM, 2018.

[92] Tom Janssen, Rui Abreu, and Arjan JC van Gemund. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE, 2009.

[93] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 298–309. ACM, 2018.

[94] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. ACM, 2017.

[95] Phil Johnson. Arg! the 9 hardest things programmers have to do. http://www.itworld.com/article/2823759/enterprise-software/124383-Arg-The-9-hardest-things-programmers-have-to-do.html#slide10, Last Accessed: August 2018.

[96] Phil Johnson. Don't go into programming if you don't have a good thesaurus. http://www.itworld.com/article/2823759/enterprise-software/124383-Arg-The-9-hardest-things-programmers-have-to-do.html, Last Accessed: Sep. 2019.

[97] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[98] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[99] René Just, Chris Parnin, Ian Drosos, and Michael D Ernst. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 287–297. ACM, 2018.

[100] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, pages 266–276. ACM, 2014.

[101] David Kawrykow and Martin P Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360. IEEE, 2011.

[102] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 295–306. IEEE, 2015.

[103] Stephen W Kent. Dynamic error remediation: A case study with null pointer exceptions. *University of Texas Master's Thesis*, 2008.

[104] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pages 802–811. IEEE, 2013.

[105] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310. ACM, 2011.

[106] Kisub Kim, Dongsun Kim, Tegawende F Bissyande, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy–a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018.

[107] Sunghun Kim and E James Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 3rd international workshop on Mining software repositories*, pages 173–174. ACM, 2006.

[108] Suntae Kim and Dongsun Kim. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering*, 21(2):565–604, 2016.

[109] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1746–1751. ACL, 2014.

[110] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2019.

[111] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2017.

[112] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. D&c: A divide-and-conquer approach to ir-based bug localization. *arXiv*, 2019.

[113] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 2019.

[114] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: bug report driven program repair. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325. ACM, 2019.

[115] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[116] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In *Proceedings of the 33nd International Conference on Machine Learning*, pages 1378–1387. JMLR.org, 2016.

[117] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 957–966. JMLR.org, 2015.

[118] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 139–148. IEEE, 2006.

[119] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Proceedings of the 14th International Conference on Program Comprehension*, pages 3–12. IEEE, 2006.

[120] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning*, pages 1188–1196. JMLR.org, 2014.

[121] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, and Shanping Li. On reliability of patch correctness assessment. In *Proceedings of the 41th International Conference on Software Engineering*, pages 524–535, 2019.

[122] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM, 2017.

[123] Xuan-Bach D Le, Quang Loc Le, David Lo, and Claire Le Goues. Enhancing automated program repair with deductive verification. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, pages 428–432. IEEE, 2016.

[124] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 213–224. IEEE, 2016.

[125] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, pages 1–27, 2018.

[126] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, pages 3–13. IEEE, 2012.

[127] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[128] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[129] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 2019.

[130] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[131] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. AntMiner: mining more bugs by reducing noise interference. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*, pages 333–344. ACM, 2016.

[132] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154. ACM, 2003.

[133] Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*, pages 53–67. Citeseer, 2006.

[134] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26. ACM, 2005.

[135] LIP6. Coccinelle. `http://coccinelle.lip6.fr/`, Last Accessed: Nov. 2017.

[136] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[137] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[138] Kui Liu, Dongsun Kim, Tegawendé François D Assise Bissyande, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, pages 1–12. IEEE, 2019.

[139] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. A closer look at real-world patches. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, pages 275–286. IEEE, 2018.

[140] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, pages 102–113. IEEE, 2019.

[141] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, pages 1–12. IEEE, 2019.

[142] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42. ACM, 2019.

[143] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*, pages 658–662. IEEE, 2018.

[144] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 118–129. IEEE, 2018.

[145] LLVM. Clang static analyzer. `https://clang-analyzer.llvm.org/`, Last Access: Nov. 2017.

[146] Rick Lockridge. Will bugs scare off users of new windows 2000, 2000.

[147] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739. ACM, 2017.

[148] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[149] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713. IEEE, 2016.

[150] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312. ACM, 2016.

[151] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 49, pages 227–238. ACM, 2014.

[152] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, pages 468–478. IEEE, 2019.

[153] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

[154] Daniel Marjamäki. Cppcheck. `http://cppcheck.sourceforge.net/`, Last Accessed: Nov. 2017.

[155] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[156] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[157] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[158] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[159] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 65–86. Springer, 2018.

[160] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Proceedings of the 36th International Conference on Software Engineering-Companion*, pages 492–495. ACM, 2014.

[161] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks*, 16(5-6):555–559, 2003.

[162] Steve McConnell. *Code complete*. Pearson Education, 2004.

[163] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 129–139. ACM, 2018.

[164] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*, pages 448–458. IEEE, 2015.

[165] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[166] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 392–402. ACM, 2015.

[167] Na Meng, Miryung Kim, and Kathryn S McKinley. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

[168] Na Meng, Miryung Kim, and Kathryn S McKinley. LASE: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. ACM, 2013.

[169] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[170] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119, 2013.

[171] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 120–130. IEEE, 2000.

[172] Naouel Moha, Yann-Gael Gueheneuc, Anne-Fran Duchien, et al. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[173] Martin Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[174] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24, 2018.

[175] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering*, 23(5):2901–2947, 2018.

[176] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1287–1293. AAAI, 2016.

[177] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[178] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 180–190. IEEE, 2013.

[179] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pages 772–781. IEEE, 2013.

[180] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring api embedding for api usages and applications. In *Proceedings of the 39th International Conference on Software Engineering*, pages 438–449. IEEE, 2017.

[181] NIST. Software errors cost u.s. economy $59.5 billion annually. `http://www.abeacha.com/NIST_press_release_bugs_cost.htm`, Last Accessed: Mar. 2018.

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

[183] ObjectLab. `http://www.objectlab.co.uk/`, Last Accessed: Nov. 2017.

[184] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, volume 42, pages 247–260. ACM, 2008.

[185] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[186] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[187] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE, 2017.

[188] Dan Pelleg, Andrew W Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the International Conference on Machine Learning*, volume 1, pages 727–734, 2000.

[189] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[190] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*, pages 654–664. IEEE, 2017.

[191] Michael Pradel and Koushik Sen. Deepbugs: a learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):147:1–147:25, 2018.

[192] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[193] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 22nd International Symposium on Software Testing and Analysis*, pages 191–201. ACM, 2013.

[194] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[195] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should I trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM, 2016.

[196] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. ACM, 2017.

[197] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806*, 2018.

[198] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories*, pages 10–13. IEEE, 2018.

[199] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. IEEE, 2017.

[200] Robert C Seacord, Daniel Plakosh, and Grace A Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[201] Haihao Shen, Jianhong Fang, and Jianjun Zhao. Efindbugs: Effective error ranking for findbugs. In *Proceedings of 4th International Conference on Software Testing, Verification and Validation*, pages 299–308. IEEE, 2011.

[202] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[203] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 130–140. IEEE, 2018.

[204] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[205] Mauricio Soto and Claire Le Goues. Using a probabilistic model to predict bug fixes. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 221–231. IEEE, 2018.

[206] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 International workshop on Mining Software Repositories*, pages 133–136. ACM, 2006.

[207] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 101–110. ACM, 2011.

[208] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 22nd International Symposium on Software Testing and Analysis*, pages 314–324. ACM, 2013.

[209] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: detecting similarly behaving software. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 702–714. ACM, 2016.

[210] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. An approach for evaluating and suggesting method names using n-gram models. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 271–274. ACM, 2014.

[211] Synopsys. Coverity. `http://www.coverity.com/`, Last Accessed: Nov.2017.

[212] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[213] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 727–738. ACM, 2016.

[214] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1422–1432. ACL, 2015.

[215] Duyu Tang, Bing Qin, and Ting Liu. Learning semantic representations of users and products for document level sentiment classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1014–1023. ACL, 2015.

[216] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74. ACM, 2014.

[217] Andreas Thies and Christian Roth. Recommending rename refactorings. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 1–5. ACM, 2010.

[218] Ferdian Thung, David Lo, Lingxiao Jiang, et al. Are faults localizable? In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 74–77. IEEE, 2012.

[219] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837. ACM, 2018.

[220] Radhika D Venkatasubramanyam and Shrinath Gupta. An automated approach to detect violations with high confidence in incremental code using a learning system. In *Proceedings of the 36th International Conference on Software Engineering - Companion*, pages 472–475. ACM, 2014.

[221] Peng Wang, Jiaming Xu, Bo Xu, Chenglin Liu, Heng Zhang, Fangyuan Wang, and Hongwei Hao. Semantic clustering and convolutional neural network for short text categorization. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, volume 2, pages 352–357, 2015.

[222] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. IEEE, 2016.

[223] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 61–72. ACM, 2010.

[224] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

[225] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172*, 2017.

[226] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.

[227] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 479–490. IEEE, 2019.

[228] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[229] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. Towards universal paraphrastic sentence embeddings. *arXiv preprint arXiv:1511.08198*, 2015.

[230] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[231] W Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[232] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[233] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[234] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[235] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. *CoRR*, 2019.

[236] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[237] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670. IEEE Press, 2017.

[238] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 789–799. ACM, 2018.

[239] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pages 416–426. IEEE, 2017.

[240] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[241] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, pages 191–200. IEEE, 2014.

[242] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63. ACM, 2014.

[243] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pages 242–251. IEEE, 2013.

[244] Deheng Yang, Yuhua Qi, and Xiaoguang Mao. An empirical study on the usage of fault localization in automated program repair. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, pages 504–508. IEEE, 2017.

[245] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[246] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static c analyzer. *Information Processing Letters*, 102(2-3):118–123, 2007.

[247] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36. ACM, 2011.

[248] Jongwon Yoon, Minsik Jin, and Yungbum Jung. Reducing false alarms from an industrial-strength static analyzer by svm. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference*, volume 2, pages 3–6. IEEE, 2014.

[249] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[250] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[251] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281. ACM, 2006.

[252] Zhenyu Zhang, Wing Kwong Chan, TH Tse, Yuen-Tak Yu, and Peifeng Hu. Non-parametric statistical fault localization. *Journal of Systems and Software*, 84(6):885–905, 2011.

[253] Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

[254] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering*, pages 913–923. IEEE, 2015.