Improving the Correctness of Automated Program Repair

by

Jinqiu Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner
Name:
Dr. Shing-Chi Cheung
Title:
Professor, Department of Computer Science and Engineering
Hong Kong University of Science and Technology

Supervisor(s)
Name:
Dr. Lin Tan
Title:
Associate Professor, Department of Electrical and Computer Engineering
University of Waterloo

Internal Member
Name:
Dr. Ladan Tahvildari
Title:
Associate Professor, Department of Electrical and Computer Engineering
University of Waterloo

Internal Member
Name:
Dr. Derek Rayside
Title:
Associate Professor, Department of Electrical and Computer Engineering
University of Waterloo

Internal-external Member
Name:
Dr. Peter Buhr
Title:
Associate Professor, Department of Computer Science
University of Waterloo

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

While I am the leading author of all the work presented in this thesis, the work is indeed the result of a group effort. In all the work of this thesis, my contributions are: drafting the initial research idea; researching background knowledge and related work; implementing the tools; conducting experiments; and writing and polishing the writing. In general, my co-authors supported me in refining the initial ideas, pointing me to missing related work, providing feedback on earlier drafts, and polishing the writing. Particularly, in Chapter 3, my co-authors also supported me in distributing the testing and running environment in multiple machines, and reproducing previous work for comparison (i.e., reproducing SPR patches). In Chapter 4, my co-authors also support me in crafting the evaluation data set of bugs, running previous work for comparison (i.e., RSRepair). In Chapter 5, my co-authors also supported me in providing valuable annotation data for evaluation.

The co-authors of all the work in this thesis are listed below:

1. *Better Test Cases for Better Automated Program Repair (Chapter 3)*
   Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, Lin Tan. In the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2017.

2. *APARE: Automatically Learning Fix Patterns from Past Fixes to Generate Recurring Fixes (Chapter 4)*
   Jinqiu Yang, Quinn Hanam, Mike Chong, Nasir Ali, Taiyue Liu, and Lin Tan.

3. *Priv: Prioritizing, Visualizing and Fixing Vulnerability Findings of Static Application Security Testing (Chapter 5)*
   Jinqiu Yang, Lin Tan, John Peyton, and Kristofer A Duer.

Abstract

Developers spend much of their time fixing bugs in software programs. Automated program repair (APR) techniques aim to alleviate the burden of bug fixing from developers by generating patches at the source-code level. Recently, Generate-and-Validate (G&V) APR techniques show great potential to repair general bugs in real-world applications. Recent evaluations show that G&V techniques repair 8–17.7% of the collected bugs from mature Java or C open-source projects. Despite the promising results, G&V techniques may generate many incorrect patches and are not able to repair every single bug.

This thesis makes contributions to improve the correctness of APR by improving the quality assurance of the automatically-generated patches and generating more correct patches by leveraging human knowledge. First, this thesis investigates whether improving the test-suite-based validation can precisely identify incorrect patches that are generated by G&V, and whether it can help G&V generate more correct patches. The result of this investigation, *Opad*, which combines new fuzz-generated test cases and additional oracles (i.e., memory oracles), is proposed to identify incorrect patches and help G&V repair more bugs correctly. The evaluation of *Opad* shows that the improved test-suite-based validation identifies 75.2% incorrect patches from G&V techniques. With the integration of *Opad*, SPR, one of the most promising G&V techniques, repairs one additional bug.

Second, this thesis proposes novel APR techniques to repair more bugs correctly, by leveraging human knowledge. Thus, APR techniques can repair new types of bugs that are not currently targeted by G&V APR techniques. Human knowledge in bug-fixing activities is noted in the forms such as commits of bug fixes, developers' expertise, and documentation pages. Two techniques (*APARE* and *Priv*) are proposed to target two types of defects respectively: project-specific recurring bugs and vulnerability warnings by static analysis.

APARE automatically learns fix patterns from historical bug fixes (i.e., originally crafted by developers), utilizes spectrum-based fault-localization technique to identify highly-likely faulty methods, and applies the learned fix patterns to generate patches for developers to review. The key innovation of *APARE* is to utilize a percentage semantic-aware matching algorithm between fix patterns and faulty locations. For the 20 recurring bugs, *APARE* generates 34 method fixes, 24 of which (70.6%) are correct; 83.3% (20 out of 24) are identical to the fixes generated by developers. In addition, *APARE* complements current repair systems by generating 20 high-quality method fixes that RSRepair and PAR cannot generate.

Priv is a multi-stage remediation system specifically designed for static-analysis security-testing (SAST) techniques. The prototype is built and evaluated on a commercial SAST

product. The first stage of *Priv* is to prioritize workloads of fixing vulnerability warnings based on shared fix locations. The likely fix locations are suggested based on a set of rules. The rules are concluded and developed through the collaboration with two security experts. The second stage of *Priv* provides additional essential information for improving the efficiency of diagnosis and fixing. *Priv* offers two types of additional information: identifying true database/attribute-related warnings, and providing customized fix suggestions per warning. The evaluation shows that *Priv* suggests identical fix locations to the ones suggested by developers for 50–100% of the evaluated vulnerability findings. *Priv* identifies up to 2170 actionable vulnerability findings for the evaluated six projects. The manual examination confirms that *Priv* can generate patches of high-quality for many of the evaluated vulnerability warnings.

Acknowledgements

I would like to take this opportunity to express my thanks to my supervisor Prof. Lin Tan. From her, I received numerous support and encouragement. I would like to thank her for insightful guidance and unwavering support. She is always available when I ask for help or advice. I learned a lot from Lin, including how to conducting research, how to shape research ideas, how to utilize time management for multi-tasking, presentations skills, and how to communicate and collaborate with other people. What I learnt from Lin will benefit my future career. I am also thankful to all of my committee members: Prof. Ladan Tahvildari, Prof. Derek Rayside, Prof. Peter Buhr, and Prof. Shing-Chi Cheung for spending their valuable time in reviewing my thesis and providing valuable comments. Also, I would like to thank Prof. Reid Holmes for providing valuable feedback on the early stage of my PhD work.

I would like to thank all of my co-authors and collaborators: Alex Zhikhartsev, Yuefei Liu, Quinn Hanam, Mike Chong, Nasir Ali, and Taiyue Liu from our research group, also John Peyton, and Kristofer A Duer from IBM AppScan Source. I would like to thank them for their support and help along my Ph.D. journey. We have shared many memories about working non-stop days and nights before conference deadlines. I would not have made these deadlines without their hard working and generous help.

My thanks also go to all the members in our research group. Many thanks for the happy time in team building activities and dinners, in our reading group seminars, and tremendous inspirations and knowledge I got from our discussions.

Last but not the least, I deeply thank my parents for their unconditional love and support for me to pursue my dream. Thanks also go to my best friend and partner, Dr. Tsehsun Chen, who has shared my happiness and frustration along the way. From them, I perceive the power of love which have been and will be with me.

Dedication

This is dedicated to my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software bugs affect software reliability and security. Developers spend much of their time fixing bugs in software programs [66]. Unfortunately, the number of bugs that need to be fixed is significantly larger than time and resources allow [13]. Over the last decade, automated program repair (*APR*) has been an emerging research area. Many APR techniques have been proposed [70, 83, 115, 51, 61, 58, 75]. The promising results of the state-of-the-art test-suite-based APR techniques (also known as Generate-and-Validate G&V) shed light on alleviating the burden of bug fixing from developers. Recent evaluations [145, 94] show that G&V techniques repair 8–17.7% of the bugs from C or Java projects. Given a set of passing test cases and at least one failing one, G&V techniques locate likely correct patches in a constructed search space of possible patches and identify the patch that passes the validation of the given test cases (i.e., makes all the test cases pass including, the failing test cases).

However, two primary limitations prohibit the advances of G&V APR techniques. First, the patch validation does not fully guarantee the correctness of the patches. Many of the patches (e.g., up to 98% as shown in [117]) that pass the patch validation are actually incorrect. Such incorrect patches are referred to as *overfitted* patches. Second, the hypothesized space of patches does not contain correct fixes for all bugs. Hence, for such cases, G&V APR techniques cannot generate a correct fix.

The above-mentioned two limitations affect the correctness of G&V APR techniques. On one hand, the patch validation (i.e., test cases) for verifying the correctness of patches

should be enhanced so the improved validation can identify the overfitted patches. On the other hand, the hypothesized space of patches should be enriched to contain more correct patches.

1.2 Background on Automated Program Repair

Automated Program Repair (APR) techniques perform "the transformation of an unexpected behavior of a program execution into an acceptable one according to an specification" [101]. APR relies on specifications to generate and/or validate repairs. Specifications can be obtained and expressed in different ways, such as test suites, behavior models, etc. Monperrus [101] presents a comprehensive bibliography on APR, which classifies APR based on the types of specifications. This section describes two types of APR in details since they are closely related to this thesis: (1) test-suite-based APR (i.e., G&V) techniques which rely on test cases for patch generation and validation; and (2) static-analysis-based APR.

G&V techniques (e.g., GenProg [70], Kali [117], AE [139], SPR [83], Angelix [94]) automatically generate patches when provided with a buggy version, both failing and passing test cases. Then, G&V techniques utilize spectrum-based fault localization techniques to narrow down the scope of the faulty source code. After that, G&V techniques use specific approaches to construct a search space of patch candidates: (1) applying modification operators (add, delete, and mutate) on the code (GenProg [70], RSRepair [116] and Kali [117]), (2) using pre-defined or automatically-learnt common fix patterns to create a repair (PAR [61], SPR [83], and Genesis [82]), and/or (3) using constraint solving to fix defective conditions (SPR [83], Angelix [94], SemFix [107], Qlose [31]). After constructing the search space, G&V techniques iterate the patches in the search space until they find a patch that can pass the patch validation (i.e., whether the patch can make the *same set of test cases* pass). Recent advances in automated program repair leverage past fixes to rank the patch candidates in the search space. For example, Prophet [85] ranks the patch candidates in SPR's search space based on a probability model that is trained from past fixes. Le et al. [68] propose (referred as *HistoryDriven*) to rank and select the top patch candidates based on the code similarity between the patch candidates and mined frequent cross-project fix patterns.

Despite their differences, G&V techniques share the same techniques of fault localization and patch validation. Imperfect patch validation (e.g., using the *same test cases* for both patch generation and validation) may lead to overfitted patches [127]. The generated patch is either a correct patch (i.e., it indeed fixes the target bug) or it is an overfitted patch (the test cases pass however the target bug is not fixed).

Static-analysis-based APR techniques generate repairs to fix the warnings by static-analysis bug detection techniques. Many techniques of this type use fix templates that are defined for each bug type detected by static analysis [80, 81, 38, 102]. For example, Gao et al. [38] automatically fix memory leaks by inserting deallocation statements. Muntean et al. [102] use parameterized fix templates to generate fixes for detected buffer overflows.

1.3 Thesis Statement

Patch generation and patch validation are two primary components in automated program repair. This thesis shows that:

> The correctness of automated program repair can be improved by enhancing the patch validation process, and also by enriching the hypothesized space of patches.

The goal of this thesis is to improve the correctness of automated program repair. First, insufficient patch validation makes G&V techniques generate incorrect patches. Hence, how to improve patch validation (i.e., obtaining better test cases) towards generating more correct patches should be explored to improve APR. The explored approach to improve the patch validation should be scalable and applicable to a wide spectrum of software. The proposed approach should be fully automated to be integrated with APR techniques.

Second, to generate more correct patches, new APR techniques should be explored to mitigate the limitation of space of patches. Current APR can only fix limited number of bugs due to the limited space of patches. Enriching the space of patches with more fix patterns allows APR generating more correct patches. Also, fix patterns of high-quality can reduce the possibility of generating incorrect patches [134]. The proposed APR techniques should complement the state-of-the-art APR by fixing previously-unhandled bugs. I believe that knowledge of how developers fix past bugs should be utilized to achieve this objective.

1.4 Thesis Contributions

This thesis improves the correctness of automated program repair from two aspects. First, it takes a first step to explore the direction of improving the test-suite-based validation of G&V techniques. Given a G&V technique, the improved test-suite-based validation can

identify overfitted fixes, and thus allow the G&V technique to generate a correct patch. *Opad* is proposed to improve the test-suite-based validation, which combines automated test generation, two oracles (crash and memory-safety), and a novel overfitness metric to detect overfitted patches. Particularly, *Opad* improves existing test suites to better define bugs and preserve the desired functionalities from two angles: (i) generating new test cases automatically, and (ii) leveraging additional oracles (i.e., memory-safety oracles) to improve validity checking of automatically-generated patches. The evaluation of *Opad* shows that a significant portion (75.2%) of overfitted patches are filtered out by *Priv*. Also, the evaluation shows that with the help of *Opad*, SPR, one of the state-of-the-art G&V techniques, generates the correct patch for one additional bug (vanilla SPR fixes 11 bugs correctly).

The second contribution this thesis makes is to propose novel repair techniques which enrich the hypothesized space of patches. In particular, the two proposed techniques utilize the knowledge of how developers fix past bugs to obtain correct patches to complement current APR techniques. The knowledge of how developers fix bugs is noted in the forms such as bug fixing commits, document/tutorials that shows how to fix bugs, and Q&A websites such as StackOverflow. Automated-extracted or manually-defined fix patterns from such materials can be used to generate patches for new bugs. Obtaining a space of high-quality patches based on developers' past bug-fixing activities reduces the possibility of producing overfitted patches. As shown by Tan et al. [134], a few patch anti-patterns (i.e., patterns that are unlikely constructed by developers) can remove a significant portion of overfitted patches.

APARE and *Priv* target bugs that are not handled well by current APR techniques. Also *APARE* and *Priv* enrich the hypothesized space of patches by adding high-quality fix patterns. *APARE*, a history-based technique that targets project-specific recurring bugs, extracts fix patterns from the history of a project (i.e., past fix commits), and applies the learnt fix patterns on buggy methods to generate recurring fixes. *APARE* utilizes past fixes which are considered as high-quality patches crafted by programmers. An evaluation on five real-world applications shows that *APARE* complements current G&V techniques by generating 20 correct method fixes that are project-specific recurring. *APARE* achieves a precision of 70.6% on the evaluated 20 recurring bugs.

Priv, a multi-phase remediation system, is designed to help developers improve work efficiency in fixing vulnerability warnings by static-analysis security testing (SAST). SAST, such as AppScan Source, employs static analysis to build information flows from malicious input (source) to an exploit point (sink). In the first phase, *Priv* produces a global-view visualization based on clusters of shared fix locations. *Priv* suggests fix locations based on rules that are defined for each vulnerability type. I created the sets of rules via discussions

with security experts (i.e., our collaborators from AppScan Source). Based on the suggested fix locations, *Priv* groups the vulnerability warnings into clusters. *Priv* produces an interactive visualization of the clusters of vulnerability warnings. In the second phase, *Priv* supplements essential information to help developers diagnose and fix vulnerabilities. First, *Priv* identifies true database/attribute-related warnings to reduce the false positives of SAST. Second, *Priv* provides a customized remediation page that includes a customized fix suggestion for each vulnerability warning. To generate the fix suggestion, I first created the fix templates from the fix examples in current remediation pages. Each fix template is defined for each vulnerability type. Then *Priv* utilizes such manually-defined fix templates to generate customized fix suggestions. The evaluation of *Priv* on six web applications shows that *Priv* suggests identical fix locations to the ones by developers for up to 100% of the vulnerability warnings. Moreover, *Priv* identifies true database- or attribute-related warnings, which constitute to 11.4–86.2% of all the database- or attribute-related warnings. Finally, *Priv* provides complete fix suggestions for many vulnerability warnings.

In summary, this thesis makes contributions to advance the state-of-the-art automated program repair. First, it explores the direction of improving test-suite-based validation for improving automated program repair. Second, it introduces two novel techniques to improve the space of patches (i.e., more correct patches) by adding fix patterns: automated-extracted fix patterns (*APARE*), and manually-defined fix patterns (*Priv*). Two proposed two novel techniques improve address the two defect classes that are not handled well by current repair techniques, such as project-specific recurring bugs and vulnerability warnings by static analysis.

The three major contributions are:

- explores the direction of improving test suites for improving APR (Chapter 3). *Opad*, a scalable and practical approach, is proposed to enhance existing test suites by generating new test cases and leveraging two oracles (crash and memory-safety). *Opad* is shown to improve four G&V techniques to repair real-world applications.
- two novel APR techniques–*APARE* (Chapter 4) and *Priv* (Chapter 5), that leverage developers' knowledge of bug fixing, are proposed to provide high-quality fix suggestions to developers. *APARE* targets project-specific recurring fixes and is capable of automatically extracting fix patterns from past fixes. *Priv* is designed for vulnerability warnings by static application security testing.
- an evaluation of *APARE* and *Priv* shows that leveraging the knowledge of bug-fixing in the past to enrich the space of candidate patches indeed produces high-quality fixes

and can significantly broaden the scope of fixes that can be generated by automated repair techniques.

Chapter 2

Related work

This chapter summarizes the related work. Section 2.1 describes the related work on automated program repair. Section 2.2 discusses previous work on studying fix patterns. Section 2.1 and Section 2.2 are related to all the work in this thesis. Section 2.3 summarizes research work on automated test generation, which is related to *Opad*. Section 2.4 lists related work on systematic editing and clone detection, which is related to *APARE*. Last, Section 2.5 presents related work on using static-analysis to detect bugs and vulnerabilities, which is closely related to *Priv*.

2.1 Automated Program Repair

Many automated program repair and debugging techniques have been proposed [58, 126, 135, 70, 61, 83, 85, 115, 109, 32, 116, 77, 10, 73, 72, 108, 149, 155, 107, 133, 124, 41]. Among the above-mentioned work, researchers propose various G&V techniques to generate patches at the source-code level. GenProg [70] is the pioneer work in this area, followed by Par [61], RSRepair [116], Kali [117], SPR [83], relifix [133], and Nopol [147]. The above-mentioned techniques differ from GenProg in terms of either search space and/or search algorithm. Par [61] uses hard-coded patch templates to construct search space. RSRepair [116] employs a random search algorithm instead of genetic programming (which is used by GenProg). Kali [117] uses a restricted search space—emphasizing on deleting operations and an exhaustive search strategy. SPR [83], which outperforms the previous work, constructs search space based on predefined transformation schemas and leverages a targeted search algorithm. The constructed search space contains more useful patches and provides a larger set of fix templates than that of Par [61].

As an alternative to G&V techniques, semantic-based automatic repair tools are proposed. Semantic-based automatic repair uses symbolic execution and constraint solvers to synthesize a patch that by design passes all the developer test cases. SemFix [107], SPR [83], Angelix [94], SearchRepair [60], S3 [69], Enumerative [11], CVC4 [118], and Qlose [31] leverage constraint solving to repair defective conditions. Chen et al. [25] use pre- and post-conditions to repair Java programs.

Domain-specific APR techniques are proposed to target particular types of bugs. Tortoise is proposed to repair configuration errors [140]. Tian et al. [136] repair buggy error-handling code in C. Albarghouthi et al. [9] repair unfair decision-making programs based on input data distribution. D'Antoni et al. [30] synthesize repairs for commands by learning from examples. Majahan et al. [87] target layout cross browser issues and use search-based approaches to generate repairs. Liu et al. [76] aim to generate high-quality patches to fix concurrency bugs. Static-analysis-based APR techniques generate repairs to fix the warnings by static-analysis bug detection techniques. Many techniques of this type use fix templates that are defined for each bug type detected by static analysis. Logozzo et al. [80] propose to repair .Net code based on a static analysis tool. Logozzo et al. [81] propose a repair technique that targets one type of integer arithmetic bugs. Gao et al. [38] automatically fix memory leaks by inserting deallocation statements. Muntean et al. [102] use parameterized fix templates to fix buffer overflows that are detected statically. Different from the prior work, *Priv* targets vulnerability warnings by static-analyis tools, which complement current static-analysis-based APR.

This thesis complements previous work by generating more correct patches. *Opad* filters out many incorrect patches of previous G&V techniques (e.g., GenProg and SPR. *APARE* and *Priv* use fix patterns from past fixes and documentation to fix more bugs correctly. In particular, *APARE* targets project-specific recurring fixes, which are not properly addressed by previous work given the complexity of the recurring fixes (large in size, and rich project-specific semantics). *Priv* targets vulnerabilities that are detected by static analysis, which are not handled by previous APR techniques.

Using Testing to Improve G&V Techniques. Xin et al. [143] propose techniques to guide test generation techniques to cover patches by G&V techniques with an assumption that perfect oracles are already available. *Opad* of this thesis shows that basic oracles can improve G&V techniques. Yu et al. [150] propose to generate new test input to guide G&V techniques generating correct patches, however the corresponding test oracles require manual effort. Differently, *Opad* of this thesis uses automated test generation to improve G&V techniques by filtering out overfitted patches, and then continuing G&V techniques to generate correct patches. *Opad* is fully automatic and requires no manual effort in generating test oracles. Liu et al. [78] propose a novel technique which leverages the

similarity of execution traces to heuristically determine the correctness of the generated patches by G&V techniques. *Opad* uses new test inputs and oracles directly to detect overfitted patches.

Utilizing Fix History to Improve G&V Techniques. Prophet [85], which is built upon SPR, builds a probability model from past fixes and uses the probability model to rank the patch candidates in the search space of SPR. Le et al. [68] propose to rank and select top patch candidates based on the code similarity between the patch candidate and past fixes. Le et al. [69] also propose combining constraint solving, customizing a constrained search space, and ranking based on code similarity to repair defective conditions efficiently. Qi et al. [144] improve G&V techniques by prioritizing syntax-related code. Rolim et al. [119] utilize a novel domain-specific language to extract syntactic structures from examples. Gao et al. [39] propose to automatically fix recurring crash bugs by mining crash traces from Q&A websites. The targeted recurring crash bugs are common across projects and are caused by incorrect usage of common APIs (e.g., popular frameworks and libraries). Xiong et al. [145] propose ACS that combines document analysis and a set of frequent predicates to repair defective conditions. Genesis [82] automatically infers common fix patterns for certain types of bugs. All the techniques above target recurring fixes that are common across projects, and thus they utilize mining from past fixes of a large collection of repositories (e.g., GitHub). *APARE* of this thesis targets project-specific recurring fixes, which are not covered by common fix patterns across projects. Thus, *APARE* complements existing APR techniques by specializing different types of bugs.

2.2 Studies of Fix Patterns, Recurring Fixes and Code Changes

Many studies are performed to understand the nature of bug fixes, code changes and fix patterns [105, 104, 92, 113, 89, 99, 23]. Nguyen et al. [105] study repetitiveness of code changes. Negara et al. [104] mine frequent code change patterns using data mining techniques. A recent study finds that 31–52% of tokens and 3–17% of lines in commits repeat cross commits [92]. Martinez, Duchien and Monperrus abstracted a subset of bug fix patterns from that study [113] at the AST level manually, and used the these AST-level patterns to search for matching code changes automatically [89]. Barr et al. [18] investigate whether fixes can be constructed using fix ingredients (i.e., fixes are decomposed at statement level) from past fixes, which is the assumption held behind GenProg, AE and RSRepair. This thesis, when presenting *APARE*, includes a manual study on project-specific recurring

fixes at method level. Specifically, the manual study explores the potential and challenges of adapting systematic editing techniques (e.g., SYDIT) for automated program repair.

2.3 Automated Test Generation

There exist different types of automated test generation. Random test generation techniques [112, 29] and fuzz testing tools scale to large systems, but lack of direction (i.e., targeting a specific code region). Dynamic symbolic execution [24] and concolic testing [123] tools aim to generate test cases that achieve high coverage. Search-based test generation techniques [36, 37] integrate search algorithms to guide unit test generation to achieve high coverage. All the above techniques use crashes as oracles. Alternatively, regression oracles are automatically generated [142] by recording variable values during running black-box test cases written by developers. This work uses crashes, which is a widely-accepted oracle, and memory-safety oracles to improve validity checking of patches.

2.4 Generating Systematic Edits, Refactoring, and Detecting Clones

Sydit [95] applies systematic edits learnt from one code transformation or fix to other similar locations based on context. LASE [96] extends Sydit, which learns from one transformation/fix, to learn from multiple ones and transform the shared part. Directly applying Sydit and LASE for automated bug fixing does not work. First, as discussed in Section 4.1.1, *APARE* advances Sydit and Lase by applying learned patterns to locations whose contexts are different from the given fix/transformation. The evaluation shows that *APARE* can generates 12 and 13 more method fixes than Sydit and LASE respectively for the 20 recurring bugs. Second, given a bug to fix, Sydit and LASE require developers to pinpoint past similar fixes for fix pattern learning, which is often difficult for developers to know (§4.4). Third, LASE requires at least two past fixes and learns the common patterns to address some limitations of Sydit. Multiple similar past fixes are not always available. Besides, using the common patterns from multiple fixes can lose fix information unique to a pattern.

Refactoring from modern IDEs support pre-defined transformation (e.g., bad code smells, and API migration) on source code [110, 97, 46, 106]. Automated software transplantation [19] is proposed to automate the process of migrating features from one software to

another. Differently, *APARE* generates method-level fixes automatically without prior knowledge of what code to be applied. Researchers have spent much effort on detecting clone code, e.g., token based [71, 59] , AST based [50], semantic based [20] and deep-learning based [141]. Clone detection [71, 59, 50, 20] tools suggest other similar locations for bug fixes. However, code clone detection techniques focus on detection of clones and do not generate fixes.

2.5 Detecting Vulnerability Warnings Using Static Analysis

Static analysis is shown to be effective in detecting vulnerabilities, especially information-flow-related vulnerabilities [121, 79, 54], such as cross-site scriptings and SQL injections. Also, static analysis is widely used to improve mobile security by detecting data leaks to protect sensitive and confidential information [16]. However, a previous study by Johnson et al. [53] shows that developers do not fully utilize static analysis techniques because of two reasons: high false positive rates and the presentation of the detected results. Chen et al. [26] discuss the experience of promoting static performance anti-pattern detection tool among developers.

To improve the usability of static analysis techniques, researchers have been working on the direction of reducing the false positive rate. Junker et al. [56] convert static analysis to a model checking problem and then utilize SMT solver to check path feasibility. Fehnker et al. [34] propose a technique to reduce false positives by leveraging refined security rules. Muske et al. [103] propose a partitioning approach to reduce false positives. The results of the static analysis techniques are divided into equivalence classes. If the leader of a class is determined as false positive, then the entire equivalence class is false positives.

In addition, many statistic-based approaches are proposed to identify false positives. Shen et al. [125] improve the FindBugs [**?**] tool by using an error-ranking strategy to increase true positive rate. Jung et al. [55] combine statistical analysis with domain knowledge to reduce false alarms by static analysis. Kremenek et al. [65] rank warnings by static analysis combining correlation between warnings and feedbacks from developers. Hallem [43] propose a flexible, easy-to-use extension language allowing users to specify system-specific rules for detecting bugs statically. Tripp et al. [**?**] extract features and interactively reduce false positives by using classification. Similarly, Hanam et al. [45] use classification based on features from code patterns and leverage past unactionable alerts (i.e., false positives) as a training set. Ruthruff et al. [120] use code metrics and false warning patterns to

predict actionable warnings. In addition, techniques are proposed to group related warnings together to reduce inspection effort in redundant warnings. Le et al. [67] compute path dependencies of warnings to group relevant warnings together to reduce redundancy.

Priv is different from the previous studies on reducing static analysis false positives and manual inspection efforts on the detected problems. *Priv* focuses on prioritizing detected vulnerability warnings using both fix location suggestion and visualization, and providing automatically generated fix suggestions for each detected vulnerability. In addition, *Priv* significantly reduces false positive rate (i.e., reduces up to 88.6%) of database-related and attribute-related vulnerability warnings, which are a different type of false positives that are not addressed in previous studies.

Chapter 3

Opad: Better Test Cases for Better Automated Program Repair

3.1 Introduction

Automated generate-and-validate program repair techniques [70, 116, 61, 133, 117, 83] (G&V techniques) show promising results to reduce manual quality assurance efforts and to improve software reliability. G&V techniques automatically generate patches to repair buggy programs with the guidance of test cases and validate the correctness of the generated patches using the same set of test cases.

Despite the great potential, G&V techniques suffer from generating incorrect patches due to in-capabilities of test suites [117, 83, 127]. Qi et al. [117] pointed out that 98% of the patches that are generated by GenProg [70] are incorrect. A large portion of such incorrect patches are equivalent to deletion of buggy functionalities. These incorrect patches make test cases pass after the entire buggy code is removed, simply because the test cases do not cover the expected correct behaviors of the buggy code. For example, for the bug libtiff-08603-1ba75 (an arithmetic bug), GenProg generates incorrect patches that remove an integer overflow check to make these given test cases pass because they do not expose the integer overflows. Following previous work [127], such incorrect patches are referred to as **overfitted patches**, since they are overfitted to pass only the given tests, but fail to fix the bugs.

Overfitted patches prevent G&V techniques from generating correct patches. The terminating condition of G&V techniques is to make all given test cases pass; thus, once an

overfitted patch is generated, G&V techniques often stop exploring other patch candidates. This happens when the failing test case cannot well define the bug and/or if the original passing test cases fail to define all correct behaviors of the software. Thus, improve test cases should be improved to be able to precisely decide whether the generated patches overfit and make G&V techniques continue to generate correct patches.

There are limited prior efforts to enhance test cases for large and complex systems to further improve G&V techniques. Previous studies [133, 127] focus on illustrating the impacts of low-quality test suites on the quality of automatically-generated patches. Recent work [150, 143] on this direction demonstrates the challenges of using automated test generation to improve G&V techniques in a real-world setting. Xin et al. [143] design a new test generation technique to cover the generated patches by G&V techniques. However, it requires correctly-patched programs to get oracles (e.g., expected outputs), which is difficult to obtain in practice.

In this work, an *O*verfitted *PA*tch *D*etection framework, *Opad*, is proposed. *Opad* combines automated test generation, two oracles (crash and memory-safety), and a novel overfitness metric to detect overfitted patches. First, *Opad* improves existing test suites to better define bugs and preserve the desired functionalities from two angles: (1) generating new test cases automatically, and (2) leveraging additional oracles (i.e., memory-safety oracles)

The approach of *Opad* is analogous to the treatment of sickness: to determine if a patient has recovered (analogous to whether a bug has been fixed by a patch), in addition to checking if symptoms have been improved, doctors often (1) order laboratory tests such as blood tests (analogous to generating new tests), and (2) check if medical metrics such as white blood cell counts have been improved compared to those when a patient is sick (analogous to the improved validity checking).

Second, *Opad* leverages a novel metric, *the overfitness measure, O-measure* in short, to assist the improved test suite in detecting overfitted patches. The proposed *O-measure* is shown to be an effective approximation of the ideal metric that can best distinguish a correct patch from an overfitted patch. A prior study [127] shows that deciding whether a patch is overfitted using whether the patched version fails on any of the additional tests is imprecise in distinguishing overfitted from correct patches. Different from this prior study, the O-measure is built based on the assumption that a correctly patched program should not behave worse than the corresponding buggy program (e.g., fail on more test cases).

Opad is applied to improve four G&V techniques, GenProg [70], AE [139], Kali [117], and SPR [83], in generating patches for 45 bugs. *Opad* automatically generates between 452 to 31,904 new test cases per bug, which include both passing and failing test cases.

The evaluation shows that:

- *Opad* filters out a significant portion (75.2%, 321/427) of overfitted patches generated by the four G&V techniques. With *Opad*, GenProg/AE, Kali, and SPR generate correct patches for 2, 3, and 12 bugs respectively.
- By filtering out overfitted patches, *Opad* helps SPR [83] generate a correct patch for one additional bug (libtiff-d13be-ccadf) compared to the original SPR (vanilla SPR generates correct patches for 11 bugs). In other words, without our approach, SPR fails to generate this correct patch. Although many overfitted patches are filtered out, *Opad* does not *always* lead to the generation of more correct patches, since, (1) to generate the correct patch, *all* overfitted patches that precede the correct one in the search space must be filtered, and (2) the search space must contain the correct patch.

3.2 The Main Contributions of this Chapter

In summary, this chapter makes the following contributions:

- enhance test suites for improving G&V techniques.
- formulate the ideal and theoretical metric for determining if a generated patch is overfitted, and propose a novel practical metric for it.
- explore and identify a scalable and practical approach to enhance existing test suites by generating new test cases and leveraging two oracles (crash and memory-safety).
- evaluate the proposed approach by applying it to improve four G&V techniques to repair large and complex systems.

3.3 Approach

Overview. Figure 3.1 shows an overview of the proposed *O*verfitted *PA*tch *D*etection framework (*Opad*) for validating the correctness of automatically-generated patches. It also shows how *Opad* can be used to improve G&V techniques. *Opad* employs automatic test generation, two test oracles (crash and memory safety), and a metric (Overfitness-measure:

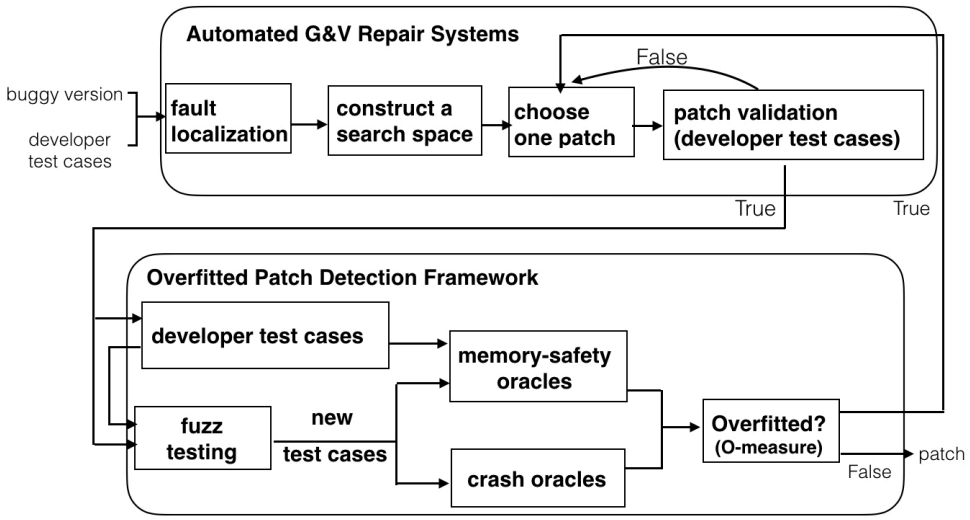

Figure 3.1: Overview of the Proposed Overfitted Patch Detection Framework (*Opad*) and How *Opad* is Integrated with G&V Techniques.

O-measure) to assess the correctness of automatically-generated patches. First, to generate new test cases, *Opad* leverages fuzz testing and uses existing test suites as fuzzing seeds. Second, for all test cases (including automatically-generated test cases and developers' original test cases), *Opad* employs two additional oracles, a crash and a memory-safety oracle (e.g., buffer overflows, uninitialized variables, and memory leaks), to improve validity checking of automatically-generated patches. Third, based on the validity results, for each automatically-generated patch, *Opad* uses *O-measure* to decide whether a patch is overfitted. Figure 3.1 shows how *Opad* complements G&V techniques by deciding whether a generated patch is overfitted. *Opad* guides G&V techniques to continue choosing the next patch candidate in the search space if a patch is identified as overfitted.

Challenges. There are two main challenges in designing an overfitted patch detection framework based on automatically-generated new test cases for large and complex systems. The first challenge is how to leverage the generated tests and bug detection tools to determine if a patch is overfitted. A naive approach is that if a patch causes any automatically-generated test to fail the improved validity checking (e.g., the patched version contains a memory bug as reported by a bug detection tool), then we consider the patch overfitted. However, this approach is likely to filter out correct patches, because there are other irrelevant bugs (i.e., bugs are not related to the target bug) in the program. A correct patch may correctly fix the target bug, but fail to fix other *irrelevant* bugs in the program, i.e., bugs that

are not targeted by the G&V tool (current G&V approaches are designed to fix only the target bug as defined by developer failing test cases). To address such irrelevant bugs, *Opad* uses O-measure (Section 3.3.3) that only considers a patch to be overfitted if the patched program performs worse than the buggy program under the same set of tests. The assumption behind the proposed O-measure is that a correctly patched version should not behave worse than the buggy version, e.g., the patched version should not fail on the test cases on which the buggy version passes. Section 3.3.3 presents how and why we define O-measure.

The second challenge is the lack of test oracles: developer-written tests usually contain manually defined test oracles (e.g., `assert` statements that compare the expected output of a program with the actual output); however, it is an open challenge to automatically generate such test oracles [12]. To address this challenge, *Opad* leverages two oracles (crash and memory-safety) to help ensure the correctness of the patches. These two oracles are correct because programs *should not* crash under any circumstances (i.e., a crash is a definite indication of a bug in the program) and *should not* violate memory safety (e.g., memory leaks). By adding new test cases, the memory-safety oracles can guarantee memory safety of more code execution paths in the program by patched G&V techniques.

3.3.1 Generating New Test Cases Using Fuzz Testing

In order to generate new test cases, *Opad* uses fuzz testing [98]—a well-established bug-finding technique that feeds the program under test with randomly-generated input. Fuzz testing is chosen due to the following constraints when improving G&V techniques on large and complex systems. First, fuzz testing is scalable to large and complex systems (i.e., programs of millions of lines of code). Currently, many other advanced automatic test generation techniques do not work for programs of such scale. Second, fuzz testing can be applied to a wide spectrum of software (from image manipulation programs to interpreters). Finally, our benchmark consists of C programs, for which there are limited tools available; unlike other languages, there are well-established tools (e.g., Randoop [111] and EvoSuite [36] for Java). Primitive fuzzing techniques rarely find errors deep within a program's control flow because the randomly-generated input is usually rejected at early stages of error checking. To mitigate this issue, mutation-based fuzzing was proposed [7, 130]. Mutation-based fuzzers perform random mutations on well-formed input, which allows mutated input to pass initial sanity checks and trigger the bugs that lie deeper in the program. In this work, we use American Fuzz Lop (AFL) [1], a coverage-guided fuzz-testing tool, to generate new test cases for the bugs in the evaluated benchmark. AFL is a mature mutation-based fuzz-testing tool that detects significant vulnerabilities in mature C

projects [1]. AFL works by applying mutation rules on input, by selecting the new input that explores new paths (to achieve higher coverage), and by continually mutating the newly created input until all inputs are explored or AFL is terminated manually.

3.3.2 Generating Memory-Safety Oracles

Opad employs memory-safety oracles on both newly automatically-generated test cases and developer test cases to improve validity checking of automatically-generated patches. *Weak oracles* (e.g., checking only whether a program crashes) are not sufficient to guarantee program correctness, which is true for both developer test cases and automatically-generated test cases. To mitigate this, *Opad* enhances validity checking of patches by inspecting the quality of memory management and ensuring memory safety.

To validate large and complex systems, we need a practical and scalable memory-safety checker. This work chose dynamic analysis over static analysis, since static analysis tools may generate too many false positives. hence, static analysis is unsuitable for our purpose since false positives might erroneously prune overfitted patches (not due to the defect in the patch); in addition, false positives are likely to prune correct patches as well.

Opad leverages Valgrind [6] (i.e., Memcheck) for memory-safety oracles. Specifically, *Opad* applies Valgrind with each test case (i.e., either from a developer or automatically-generated) and records the detection results from Valgrind (i.e., memory errors and leaked memory bytes). Valgrind inspects memory safety by instrumenting the program under test, keeping track of validity of all unallocated/allocated memory, and reporting errors once memory safety is violated. Valgrind can detect various memory-related problems, such as using undefined values, accessing already-freed memory, and memory leaks.

3.3.3 Measuring the Overfitness of a Patch Using an Overfitness Metric (O-measure)

This section presents the definition of *O-measure* in *Opad*. *Opad* uses O-measure to determine whether automatically-generated patches overfit. Then, this section gives a justification about why the proposed definition of *O-measure* works best under both theoretical and practical constraints for G&V techniques.

Defining O-measure

O-measure is proposed to identify overfitted patches. The proposed *O-measure* is calculated based on the results of executing test cases (both developer and automatically-generated) against two oracles (crash and memory-safety).

The following presents the definition of O-measure and how to use O-measure to decide overfitness of patches below.

Definition 1. *Given a test suite T,*
B*: the set of test cases that make the buggy version fail ($B \subset T$),*
$\overline{B}$*: the set of test cases that make the buggy version pass ($\overline{B} \subset T$),*
P*: the set of test cases that make the patched version fail ($P \subset T$).*
O-measure is defined as the cardinality of $\overline{B} \cap P$.

Definition 2. *A patch is overfitted if it has a non-zero O-measure, and not overfitted otherwise.*

Calculating O-measure

Opad executes each test case on both versions (the buggy and the patched versions) and records the oracle-related execution results (i.e., whether the program crashes or there are memory-safety issues). Based on the results, *Opad* calculates O-measure to determine the overfitness of patches. If O-measure is non-zero for a patch, *Opad* determines the patch to be overfitted, and not overfitted otherwise.

It is straightforward to calculate O-measure for test cases with crash oracles. For memory-safety oracles, *Opad* decides whether a test case contributes to O-measure ($\overline{B} \cap P$) by checking whether the patched version exposes more memory issues than the buggy version. Different from crash oracles, for which the result is a binary value (whether the program crashes), memory-safety oracles produce comprehensive memory detection results. Thus, simply using whether memory safety is violated for deciding failure is not sufficient. Instead, we calculate O-measure by checking whether the patched version exposes more memory issues than the buggy version. For example, Valgrind reports memory errors (e.g., “use of uninitialized values") and the number of bytes leaked (definitely/indirectly/possibly lost). If for a test case, the patched version contains extra memory errors or extra leaked bytes of the three types above-mentioned, the value of O-measure of this patch is incremented by one.

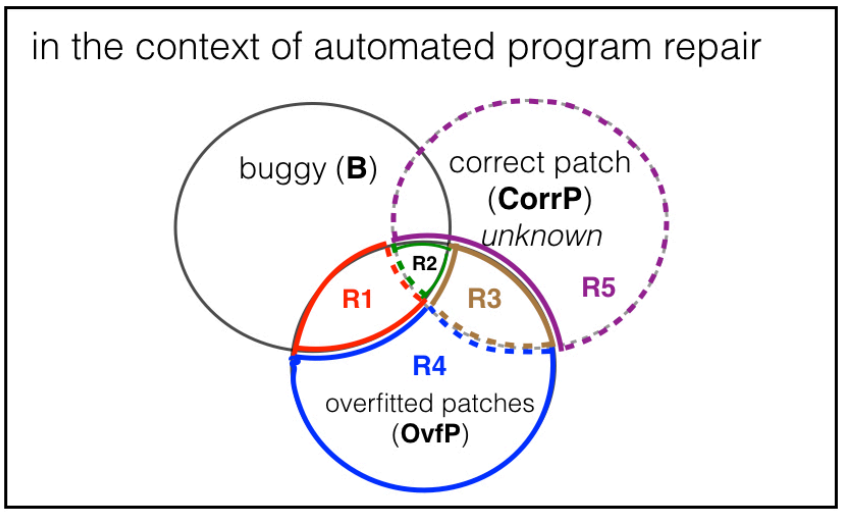

Figure 3.2: Sets of failing test cases on the buggy version (B), the versions with overfitted patches (OvfP), and the version with correct patch (CorrP).

Reasons Behind the Proposed O-measure

The proposed definition of O-measure (Definition 1) is merely one possible way to define overfitness of patches. This section illustrates why this O-measure definition is chosen from both theoretical and practical aspects.

The Ideal Overfitness Measure (O-measure). The *ideal O-measure* is defined as the O-measure that can perfectly distinguish overfitted patches from correct patches. Figure 3.2 demonstrates the relationship among the sets of failing test cases on the buggy version (annotated as B), on the correctly-patched version ($CorrP$), and on the overfittedly-patched version ($OvfP$). $\overline{B}$, $\overline{CorrP}$, and $\overline{OvfP}$ are used to annotate the sets of *passing* test cases on the buggy version, the correctly-patched version, and the overfittedly-patched version. In Figure 3.2, the five regions are highlighted: $R1$ is $B \cap OvfP \cap \overline{CorrP}$; $R2$ is $B \cap OvfP \cap CorrP$; $R3$ is $\overline{B} \cap OvfP \cap CorrP$; $R4$ is $\overline{B} \cap OvfP \cap \overline{CorrP}$; and $R5$ is $\overline{OvfP} \cap CorrP$. In addition, two regions are defined to fully understand the cardinality of all regions in Figure 3.2: $R6$ is $B \cap \overline{OvfP} \cap CorrP$, and $R7$ is $B \cap \overline{OvfP} \cap \overline{CorrP}$. Table 3.1 shows the statistics of the seven above-mentioned regions (i.e., cardinality of each region) from the data set of 45 bugs, and 429 patches from G&V techniques.

The *ideal O-measure* should be able to differentiate between correct and overfitted patches. This requirement means that there exists at least one test case that shows different

Table 3.1: The table shows the statistics of the seven regions based on 429 patches that are generated by four G&V techniques.

	R1	R2	R3	R4	R5	R6	R7
Minimum	0	0	0	9	0	0	0
Maximum	47	611	23	597	16	16	11,378
Median	0	275	3	1	1	0	0
Average	0.47	166.95	2.52	2.48	0.97	0.68	319.06
Sum	202	71,620	1,082	1,062	417	291	136,878
Zero	417	97	173	207	202	233	408
Non Zero	13	333	257	223	228	197	22

behaviors (i.e., fail or pass on the oracle) on the two versions (i.e., the one with the correct patch, and the one with an overfitted patch). So, the *ideal O-measure* for deciding overfitness is the size of the set $(OvfP \cap \overline{CorrP}) \cup (\overline{OvfP} \cap CorrP)$ ($R1 \cup R4 \cup R5$ in Figure 3.2). If the *ideal O-measure* of a patch is non-zero, this patch is overfitted as it has different behaviors from the correct patch on at least one test case.

From the Ideal O-measure to the Chosen Definition of O-measure (Definition 1). The *ideal O-measure* is annotated as $R1 \cup R4 \cup R5$ (Figure 3.2). In the context of automated program repair, the correct patch is not available, which means that $R5$ (which is a subset of $CorrP$) is hard to approximate in practice. Thus, the *first-step* approximation of the *ideal O-measure* is taken: using $R1 \cup R4$ (a subset of $OvfP$). However, $R1 \cup R4$ (a subset of $OvfP$) still cannot be directly computed due to the unavailability of $CorrP$: $R2$ and $R3$ cannot be excluded precisely. The *second-step* approximation is taken: using $R3 \cup R4$ in Figure 3.2 (our O-measure definition, Definition 1) to approximate $R1 \cup R4$ by excluding $R1$ and including $R3$. $R1$ is $B \cap OvfP \cap \overline{CorrP}$, and $R3$ is $\overline{B} \cap OvfP \cap CorrP$. The inclusion of $R3$ in inevitable to approximate $R4$ due to the unavailability of correct patch. The next paragraph illustrates why the exclusion of $R1$ is a reasonable choice in the context of using *Opad* to improve G&V techniques.

First, I prove that for a particular type of bugs and their corresponding overfitted patches, $R1$ is empty in theory. If $R1$ is empty, $R1 \cup R4$ (the first-step approximation described above) equals to $R4$. Thus, for these cases, using $R3 \cup R4$ (the defined O-measure, a superset of $R4$) identifies all overfitted patches that can be identified by the first-step approximation of the *ideal O-measure*. The proof in "*Proving the emptiness of* $B \cap OvfP \cap \overline{CorrP}$ *for specific cases*" is described below. I manually investigate how many bugs and their corresponding overfitted patches (GenProg 2012 benchmark that is used

for evaluation) fall into this particular pattern in this proof. My manual analysis shows that in theory, $R1$ is empty for 19% of the bugs (7/36, 36 bugs for which there is at least one overfitted patch from the four G&V techniques), and their corresponding 34 overfitted patches. Empirically, we find that for 97% of the patches by G&V techniques, $R1$ is empty (i.e., the cardinality is zero). Table 3.1 concludes the statistics of the cardinality of $R1$, and the average is 0.47, which is close to 0.

Second, $R1$ has to be approximated using $R1 \cup R2$ ($B \cap OvfP$). Such approximation introduces the inclusion of $R2$. Since $R2$ is part of $CorrP$ and is not part of the *ideal O-measure*, the inclusion of $R2$ causes two risks: 1) ineffectiveness in filtering out overfitted patches, especially if $R2 == B \cap OvfP$; and 2) incorrectly filtering out correct patches. Empirically, we find that both of the two risks are true in the evaluation: the bugs in the GenProg 2012 benchmark, the patches from the four G&V techniques (GenProg/AE, Kali, and SPR), and automatically-generated test cases by *Opad*. Particularly, for 92% of the overfitted patches in the evaluation, $R2 == B \cap OvfP$. This shows that using $B \cap OvfP$ to approximate $R1$ is ineffective to filter out overfitted patches since for most cases, $R1$ is empty. In addition, $B \cap CorrP$ (i.e., $R1 \cup R2$ when an automatically-generated patch is correct instead of overfitted) is not empty for correct patches of 53% of the bugs. These examples show that using $R1 \cup R2$ as O-measure or part of O-measure incorrectly filters out correct patches for 53% of the evaluated bugs. This is also confirmed by a previous work [127], which shows that using $OvfP$ as O-measure is ineffective.

In summary, *Opad* uses the proposed definition of O-measure (Definition 1) due to both theoretical and practical concerns. The intuition behind the proposed O-measure is that the patched program should not behave worse than the buggy program.

Proving the emptiness of $B \cap OvfP \cap \overline{CorrP}$ for specific cases

This proof shows that for particular types of bug and their corresponding overfitted patches, the proposed O-measure is the most reasonable metric to distinguish between correct and overfitted patches. Note that the proposed O-measure is not tied to this particular type of bug, and it also applies to other bugs (as shown in the evaluation).

We first describe the particular type of bugs and its corresponding overfitted patches, and then show that for these bugs and patches, there do not exist test cases in *R1* ($B \cap OvfP \cap \overline{CorrP}$) in Figure 3.2. First, the code structure of this particular type of bugs is:

if (`cond`) `S1`; **else** `S2`;

where `S1` and `S2` are code statements. Second, this particular type of bugs and their

corresponding overfitted patches satisfy the following conditions (which constitute 19% of the studied benchmark):

(A_1) I represents the entire input space; $I = I1 \cup I2$ and $I1 \cap I2 = \varnothing$.

(A_2) For a buggy program (B), for every input i in I, `S1` is always executed.

We use "$B(I \rightsquigarrow$ `S1`$)$" to represent that on a buggy program, `S1` is executed for every input in I. In the context of G&V techniques, there must exist at least one test case (i.e., a pair of an input i and an oracle) so that $B(i \rightsquigarrow$ `S1`$)$ leads to a failure as the oracle is not satisfied. This proof should cover all possible test cases in theory. It is unnecessary and unrealistic to obtain the result of executing every possible test case because the proof is generalizable for both cases: $B(I \rightsquigarrow$ `S1`$)$ leads to either a failure or a pass.

(A_3) Overfitted patches modify conditions to redirect every input in I to execute `S2`.

(A_4) Correct patches make the failing test case pass by redirecting every input in $I1$ to execute `S2`, while keeping every input in $I2$ to execute `S1`.

(A_5) Both overfitted and correct patches change program executions by only modifying `cond`. Thus, such patches have no side effects on other parts of the program other than that the execution flow is changed, e.g., from executing `S1` to `S2`. This means, for example, for the same input i, the results of executing $B($i $\rightsquigarrow$ `S1`$)$, $CorrP($i $\rightsquigarrow$ `S1`$)$, and $OvfP($i $\rightsquigarrow$ `S1`$)$ are the same as long as they all execute `S1`.

Proof of emptiness of $B \cap OvfP \cap \overline{CorrP}$. We start by inferring the following facts from the conditions:

(F_1) $CorrP(I1 \rightsquigarrow$ `S2`$)$.
From A_4.

(F_2) $OvfP(I1 \rightsquigarrow$ `S2`$)$.
From A_1 and A_3.

(F_3) $CorrP(I1 \rightsquigarrow$ `S2`$) == OvfP(I1 \rightsquigarrow$ `S2`$)$.
From F_1, F_2 and A_5. This means that $CorrP(I1 \rightsquigarrow$ `S2`$)$ and $OvfP(I1 \rightsquigarrow$ `S2`$)$ have the same result, i.e., either both failure or both pass.

(F_4) $CorrP(I2 \rightsquigarrow$ `S1`$)$.
From A_4.

(F_5) $B(I2 \rightsquigarrow \texttt{S1})$.
From A_1 and A_2.

(F_6) $CorrP(I2 \rightsquigarrow \texttt{S1}) == B(I2 \rightsquigarrow \texttt{S1})$.
From F_4, F_5, and A_5. Similar to F_3.

We prove by contradiction. If $B \cap OvfP \cap \overline{CorrP}$ is not empty, there exists at least one test case that satisfies all three conditions: fails on B (denoted as $Condition_1$), fails on $OvfP$ ($Condition_2$), and passes on $CorrP$ ($Condition_3$).

From A_1, the input of such test case must be either *I1* or *I2*: 1) if the input is *I1*, based on F_3, $CorrP(I1 \rightsquigarrow \texttt{S2})$ and $OvfP(I1 \rightsquigarrow \texttt{S2})$ should have the same result, either both failures or both passes. This means that $Condition_2$ and $Condition_3$ cannot be satisfied at the same time; and 2) if the input is *I2*, based on F_6, $CorrP(I2 \rightsquigarrow \texttt{S1})$ and $B(I2 \rightsquigarrow \texttt{S1})$ should have the same result, thus $Condition_1$ and $Condition_3$ cannot be satisfied at the same time.

Thus, such test case that satisfies all the three conditions does not exist, which means $B \cap OvfP \cap \overline{CorrP}$ is empty.

□

The proof above shows that the proposed O-measure is the most reasonable one for this particular type of bug. In addition, the O-measure also works well for other bugs (as shown in the evaluation).

3.3.4 An Optimized Setting of *Opad*

Opad calculates O-measure based on running test cases against test oracles. Since *Opad* uses O-measure by only asserting whether it is zero or not, *Opad* can be optimized by deciding a patch is overfitted as soon as O-measure becomes non-zero. For example, for a patch from G&V techniques, once a test case (new or developer test case) against test oracles (i.e., crash or memory-safety) fails on the patched version but not on the buggy version, *Opad* decides this patch is overfitted. Furthermore, when examining the next patch from the search space of G&V techniques, *Opad* can prioritize running the test cases with oracles that have contributed to filtering out overfitted patches before. In our evaluation, we evaluated *Opad* without this optimization to get a full understanding of the effectiveness of O-measure unless specified. However, we find that, by using this optimization, we can significantly speed up *Opad* (e.g., from over 100 to less than 10 minutes for *Opad* to guide SPR to generate a correct patch for libtiff-d13be-ccadf, a loose condition bug).

3.4 Evaluation

This section presents the experimental setup and the three research questions answered in this evaluation.

Experimental Setup. *Opad* is evaluated on the same set of bugs evaluated by previous work (GenProg, AE, Kali, and SPR). Particularly, we select all bugs for which at least one of the four repair tools have generated at least one patch. In total, *Opad* is applied on 45 bugs from 7 systems, and 449 corresponding patches (both overfitted and correct ones) that are generated by G&V techniques. To generate new test cases, we feed AFL (Section 3.3.1) with input from non-crashing developer test cases (i.e., test cases that do not make the program crash). Such non-crashing test cases include all passing test cases and some failing test cases if the failures are observed by non-crash oracles (e.g., defined expected output). The reason is that AFL, by its design, does not mutate crashing test cases in order to avoid focusing on the exact same crash. In this evaluation, AFL is terminated when no new paths are explored within two hours, since AFL may keep running without manual interruption. AFL leverages coverage to guide the mutation for better performance, and the coverage is obtained by running executables from the program under test. For some evaluated systems that contain more than one executable, AFL is only applied on the executables that are identified to expose the target bug by developer test cases. *Opad* executes each automatically-generated test case against crash oracles ten times to mitigate possible non-determinism. This number was chosen as an acceptable trade-off between efficiency of running test cases and efficacy of mitigating non-determinism. The experiment is primarily conducted in the virtual machine image released by Le Goues et al. [70], except for SPR's patches that are obtained from the SPR virtual machine [83]. We host the virtual machines on computers with 16G RAM and 3.10 GHz Intel i5 CPU.

RQ1: How many overfitted patches does *Opad* filter out?

Motivation. Identifying overfitted patches is crucial for G&V techniques since it allows them to continue exploring the search space to eventually find the correct patch. Note that it is not realistic for G&V techniques to iterate over the entire search space to find all patches that make the test cases pass. As stated in a recent study [84], there can be up to thousands of overfitted patches per search space. So, stopping at the first patch that makes all the test cases pass is a reasonable design choice for G&V techniques. Even if one generates all patches that make the test cases pass, filtering out overfitted patches could still save developers' time in selecting the correct one, as often a few correct patches are hidden among many overfitted patches [84].

Table 3.2: The results of using *Opad* to filter out overfitted patches from GenProg/AE, Kali, and SPR. 'Total' is the number of overfitted patches evaluated by *Opad*; for Kali, this number is always one (unless a Kali's patch is correct). Check symbol (✓) means that GenProg/AE, Kali, or SPR find the correct patch, and these correct patches are not incorrectly pruned by our approaches. Double check symbol (✓✓) means that *Opad* guides SPR to generate a correct patch (original SPR does not generate this correct patch).

Bug	GenProg/AE					Kali				SPR				
	Total	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All	Total	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All
gzip-3fe0-39a3	9	3	0	0	3	1	0	0	1	1	0	0	0	0
gzip-a1d3-f17c	1	0	0	0	0	-	-	-	-	1	0	0	0	0
libtiff-08603-1ba75	6	5	0	0	5	0	0	0	0	1	1	1	1	1
libtiff-5b021-3dfb3	9	9	1	9	9	1	0	1	1	238	238	0	0	238
libtiff-90d13-4c666	1	0	0	0	0	0	0	0	0	1	0	0	0	0
libtiff-d13be-ccadf	6	3	0	0	3	1	0	0	1	13	13 ✓✓	0	0	13 ✓✓
libtiff-ee2ce-b5691	1	0	0	0	0	0	0	0	0	0	0 ✓	0 ✓	0 ✓	0 ✓
lighttpd-1794-1795	10	0	0	0	0	-	-	-	-	1	0	0	0	0
lighttpd-1806-1807	6	0	0	0	0	-	-	-	-	1	0	0	0	0
lighttpd-1913-1914	1	0	0	0	0	-	-	-	-	1	0	0	0	0
lighttpd-1948-1949	-	-	-	-	-	-	-	-	-	1	0	0	0	0
lighttpd-2330-2331	9	0	2	0	2	-	-	-	-	1	0	0	0	0
lighttpd-2661-2662	9	0	0	0	0	-	-	-	-	1	0	0	0	0
python-69223-69224	1	0	0	0	0	-	-	-	-	1	0	0	0	0
python-69368-69372	-	-	-	-	-	-	-	-	-	1	0	0	1	1
python-69709-69710	-	-	-	-	-	-	-	-	-	1	0	0	0	0
python-69783-69784	3	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	1	0	0	0	0
python-70019-70023	-	-	-	-	-	-	-	-	-	1	0	0	0	0
python-70098-70101	1	-	1	0	1	0	1	0	1	1	0	0	0	0
wireshark-37112-37111	10	0	10	0	10	0	1	1	1	1	0	0	1	1
wireshark-37172-37171	1	0	1	0	1	0	0	0	0	1	0	0	0	0
wireshark-37172-37173	1	0	1	0	1	0	0	0	0	1	0	1	0	1
wireshark-37284-37285	-	-	-	-	-	-	-	-	-	1	0	0	0	0
php-307562-307561	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0	0
php-307846-307853	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0	0
php-307914-307915	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-307931-307934	9	0	0	1	1	0	0	0	0	1	0	0	0	0
php-308262-308315	-	-	-	-	-	-	-	-	-	3	0	2	0	2
php-308323-308327	-	-	-	-	-	-	-	-	-	1	0	0	1	1
php-308525-308529	1	0	0	0	0	0	0	0	0	1	0	1	1	1
php-308734-308761	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309111-309159	1	0	0	0	0	-	-	-	-	1	0	0	0	0
php-309516-309535	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309579-309580	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309688-309716	-	-	-	-	-	-	-	-	-	1	0	0	0	0
php-309892-309910	0	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309986-310009	10	0	5	0	5	0	0	0	0	1	0	1	0	1
php-310011-310050	9	0	6	5	8	0	1	0	1	6	0	5	0	5
php-310370-310389	-	-	-	-	-	0	1	0	1	1	0	0	0	0
php-310673-310681	2	0	0	0	0	0	0	0	0	1	0	0	0	0
php-310991-310999	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-311323-311300	-	-	-	-	-	-	-	-	-	1	0	0	0	0
php-311346-311348	-	-	-	-	-	0 ✓	0 ✓	0 ✓	0 ✓	0	0 ✓	0 ✓	0 ✓	0 ✓
gmp-13420-13421	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
gmp-14166-14167	3	0	0	0	0	0	0	0	0	1	0	0	0	0
Sum	120	20	27	15	49	3	4	2	7	290	252	11	5	265

Approach. *Opad* is applied on four automated G&V techniques (GenProg, AE, Kali, and SPR) to study whether our approach can correctly filter out overfitted patches while preserving correct patches. Specifically for GenProg, AE and Kali, the generated patches are publicly available. So we apply our approach on the released patches and report how many overfitted patches are successfully pruned by *Opad*. We obtained the ground truth for correct and overfitted patches from Qi et al. [117]. A patch is correct if it fixes the bug. Conversely, a patch is overfitted if it merely causes the test cases to pass and does not fix the bug. For the bugs that we evaluate, the correct patches from G&V techniques are semantically equivalent to developer patches.

Results. In total, *Opad* filters out 75.2% (321/427) overfitted patches from the four automated G&V techniques. Table 3.2 shows the overall result of filtering out overfitted patches. Table 3.2 contains all the bugs (45 in total) from the GenProg 2012 benchmark, for which at least one of the four G&V techniques can generate patches (i.e., overfitted or correct). Since AE is an adaptive version of GenProg based on a different search algorithm, we merge GenProg and AE into one column. To show the improvement from different components of *Opad*, we show the number of pruned patches in four settings: 1) using crash oracles on new test cases from fuzz testing (Column "Crash + Fuzz"); 2) using memory-safety oracles on developer test cases (Column "Mem. + Dev."); and 3) using memory-safety oracles on new test cases from fuzz testing (Column "Mem. + Fuzz"); 4) using the combination of all the above (*Opad*, Column "All").

GenProg/AE often generate several patches for a bug, so we show the total number of patches per bug in Column "GenProg/AE"/"Total". Kali generates one patch per bug, which is a total of 17 overfitted patches from Kali (we omit the column that shows the total number of patches for Kali). For some bugs that SPR has correct patches in the search space, we set SPR to continue exploring the search space until a patch is accepted by *Opad*. Therefore, the number of patches from SPR that are filtered out by *Opad* may be more than one. Column "SPR"/"Total" shows the total number of overfitted patches from SPR that are evaluated by *Opad*.

The three components of *Opad* mostly complement each other: 1) using crash oracles on fuzz test cases filters out 276 overfitted patches ("Crash+Fuzz"); 2) using memory-safety oracles on developer test cases filters out 42 overfitted patches ("Mem+Dev."); and 3) using memory-safety oracles on fuzz test cases filters out 24 overfitted patches ("Mem+Fuzz").

Opad does not filter out correct patches for ten bugs. We use '✓' or '✓✓' in Table 3.2 to annotate the bugs that *Opad* preserves the correct patches for them. However, for three other bugs, *Opad* filters out three correct patches (libtiff-5b021-3dfb3, php-307562-307561, and php-307846-307853) because the correctly-patched programs behave worse than the

```
1 int TIFFWriteDirectoryTagCheckedRational(double value, ...) {
2   assert(value >= 0.0);  // failed assertion
3 // the earliest version
4 - if (value == (uint32)value) { ...
5 - } else if (value < 1.0) {...
6 -  }
7 + if (value <= 0.0) {    // the current version
8     ...}
9   ...}
```

Figure 3.3: A bug hidden in the buggy version of libtiff-5b021-3dfb3.

buggy programs based on O-measure (i.e., either crash or fail the memory-safety oracles on some test cases, while the buggy program does not). *This happens because there are some hidden bugs in the buggy version, and such hidden bugs are exposed after the patches are applied.* Such hidden bugs are exposed in the patched version once the patch changes the control flow of the program (some of these hidden bugs are later fixed by the developers).

We show an example of a hidden bug. In libtiff-5b021-3dfb3, a hidden bug that is caused by a failed assertion is newly exposed by a correct patch (line 2 in Figure 3.3). We reported the bug to libtiff developers and the bug has been fixed[1]. This failed assertion should have been removed after the functionality had been changed. The earlier version of this function contains the assertion to abort the program if `value` is invalid. However, this function was later modified to capture an invalid `value` in the `if`-branch (line 7) and the assertion became obsolete. The buggy version exits before reaching the assertion (due to the nature of the target bug), but the correctly patched version continues the execution until the assertion fails; this results in a non-zero O-measure for the correct patch. Nonetheless, the generated test cases and our approach should help developers fix the new bugs in the patched version and, ultimately, improve the quality of the software.

RQ2: Can *Opad* guide SPR to generate correct patches for more bugs?

Motivation. We want to evaluate whether *Opad* can improve automated G&V techniques in terms of generating correct patches for more bugs. In this evaluation, we focus on SPR because SPR is shown to have great potential: there are many correct patches in the SPR search space that are not discovered because they are blocked by overfitted patches. A prior study [84] shows that there are no more correct patches in the search space of GenProg/AE and Kali to be discovered for the bugs in this evaluation. Therefore, although our approach filters out many overfitted patches generated by GenProg/AE, continuing to

[1] http://bugzilla.maptools.org/show_bug.cgi?id=2535

Table 3.3: Results of using *Opad* to improve SPR (SPR+*Opad*) on the 19 bugs from the GenProg 2012 benchmark. Each cell contains two symbols. The first symbol shows whether SPR+*Opad* generates a correct patch (**Y** or N); and the second symbol shows how *Opad* contributes in the patch generation process—✓: filtering out overfitted patches, −: not filtering out patches (neither overfitted nor correct), *B*: filtering out both overfitted and correct patches, and ×: filtering out correct patches only.

Bug ID	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All	Bug ID	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All
gzip-a1d3d-f17cb	N −	N −	N −	N −	php-309111	N −	N −	N −	N −
libtiff-5b021-3dfb3	N B	N −	N −	N B	php-309516	**Y** −	**Y** −	**Y** −	**Y** −
libtiff-d13be-ccadf	**Y** ✓	N −	N −	**Y** ✓	php-309579	**Y** −	**Y** −	**Y** −	**Y** −
libtiff-ee2ce-b5691	**Y** −	**Y** −	**Y** −	**Y** −	php-309688	N −	N −	N −	N −
python-69783-69784	N −	N −	N −	N −	php-309892	**Y** −	**Y** −	**Y** −	**Y** −
php-307562	**Y** −	**Y** −	N ×	N ×	php-310011	N −	N ✓	N −	N ✓
php-307846	**Y** −	**Y** −	N ×	N ×	php-310991	**Y** −	**Y** −	**Y** −	**Y** −
php-307914	**Y** −	**Y** −	**Y** −	**Y** −	php-311346	**Y** −	**Y** −	**Y** −	**Y** −
php-308262	N −	N ✓	N −	N ✓	gmp-13420	**Y** −	**Y** −	**Y** −	**Y** −
php-308734	**Y** −	**Y** −	**Y** −	**Y** −					

run GenProg/Kali does not generate more correct patches. In contrast, SPR has correct patches for eight more bugs in its search space (but fails to generate correct patches due to having too many overfitted patches).

Approach. *Opad* is integrated with SPR to see if *Opad* can guide SPR to generate correct patches for more bugs (see the integration in Figure 5.3). Particularly, whenever SPR generates one patch that makes existing test cases pass, that patch is validated by *Opad* (by calculating O-measure based on new test cases and two oracles). If this patch is determined as overfitted by *Opad* (O-measure is non-zero), SPR continues exploring the search space until it finds the next patch that can pass both the original validation (developer test cases) and *Opad*.

Results. *Opad* guides SPR to generate a correct patch for one additional bug (SPR previously fixed 11 bugs). Table 3.3 shows the results of applying SPR+*Opad* on the 19 bugs (from the GenProg 2012 benchmark) for which there are correct patches in SPR's search space. We use '**Y**✓' to annotate the case that, *Opad* helps SPR generate the correct patch. For libtiff-d13be-ccadf (a loose condition bug), SPR cannot generate a correct patch without *Opad*: our approach prunes 13 overfitted patches that block the correct one.

Finding the correct patch. We describe how *Opad* exposes the flaws in overfitted patches for libtiff-d13be-ccadf (a loose condition bug), filters them out and finds the correct patch. The bug is in the image-reading routine (simplified code is presented in Figure 3.4).

```
1  - if (nstrips > 1  // buggy
2  + if (nstrips > 2  // developer
3  + if (nstrips > 1 && 0  // overfitted
4      && compression == COMPRESSION_NONE
5      && stripbytecount[0] != stripbytecount[1]) {
6        TIFFWarning("Wrong␣field,␣ignoring␣and"
7          "calculating␣from␣imagelength");
8      if (estimate(tif, ...) < 0)
9        goto bad;
10 }
```

Figure 3.4: Patches for libtiff-d13be-ccadf (a loose condition bug).

The function `estimate()` at line 8 should only be called when input images are ill-formed. However, since the condition at line 1 is incorrect, `estimate()` is called for some well-formed images as well. The correct patch fixes the condition so that `estimate()` is only called when it should be. However, an overfitted patch removes the entire branch (line 3) by adding `&&0` to the condition. Thus, `estimate()` is never called, even for ill-formed images. When some automatically-generated test cases exercise the overfitted patch with ill-formed images, one garbage item, which should otherwise be cleaned up in the `estimate()` routine, is used as an array index and this leads to a segmentation violation. All the overfitted patches that precede the correct one have the described flaw and, therefore, have a non-zero O-measure and are filtered by *Opad*. Thus, with the help of automatically-generated test cases, *Opad* successfully guides SPR to generate the correct patch.

3.5 Threats to Validity

Non-determinism. Some studied programs show non-deterministic behaviors during the execution of the automatically-generated test cases. For example, a program may only crash one out of 10 times given the same input (e.g., due to address space layout randomization). To mitigate this issue, we execute each automatically-generated test case 10 times, which reduces the risk of getting spurious results and erroneously filtering out a patch.

Hidden Bugs. In some cases, a correct patch can have a non-zero O-measure because of hidden bugs. Such correct patches change the control flow of a program and reveal the bugs that were hidden in the buggy version; this leads to more crashes in the patched version compared to the buggy version and results in a non-zero O-measure (as described in Section 3.4 for libtiff-5b021-3dfb3). Although the correct patch would not be accepted by *Opad* in this scenario, the test cases that we generated would help developers fix such hidden bug manually, which remains as future work.

Limitations of Fuzz Testing. Fuzz testing has a limitation of targeting *initial* levels

of input-parsing in programs under test [93]. If a particular patch correctly fixes a bug in deeper levels of programs, but the program contains other bugs in initial levels, then the fuzz test cases would crash the program without even reaching the patched code. Our definition of O-measure eliminates this issue by a comparison with the buggy version: both versions fail on such a test case and the correct patch is not filtered out.

3.6 Chapter Summary

In conclusion, we experiment with ways to improve existing test cases in order to improve G&V techniques. We propose an approach to filter out incorrect patches by augmenting existing test cases. *Priv* improves existing test cases from two angles—better validity checking by employing memory-safety oracles and new test cases from fuzz testing. We propose O-measure, to filter out overfitted patches based on the new test cases and oracles. Our evaluation on 45 bugs from 7 systems shows that *Priv* filters out 75.2% of the overfitted patches. More importantly, *Priv* helps SPR generate the correct patch for one additional bug (original SPR can only generate correct patches for 11 bugs).

Chapter 4

APARE: Automatically Learning Fix Patterns from Past Fixes to Generate Recurring Fixes

4.1 Introduction

Developers spend much of their time fixing bugs in software programs [66]. Unfortunately, the number of bugs that need to be fixed is significantly larger than the time and resources allow [13]. Therefore, researchers have recently proposed a number of techniques that aim to automatically repair programs [70, 115, 117, 51, 61, 58, 75].

Along with proposing techniques for automated repair, researchers recently performed empirical studies on the nature of bug fixes performed by developers [90, 154, 108]. One study on bug fixes of Java programs shows that 13.3–27.7% of bug fixes are identical or similar to another fix that is committed earlier within the same project [108]. This observation suggests that within a project, past fixes may be leveraged to automatically fix future bugs. Fixes that are identical or similar to a past fix are referred to as *recurring fixes*. This work limits recurring fixes to *project-specific recurring fixes* and not target the general recurring fixes across projects (e.g., adding a null check for null pointer exception).

For example, Fig. 4.1 (automatically generated by our approach) and Fig. 4.2 show two similar fixes from Eclipse SWT. "-" denotes a line to be deleted, and "+" denotes a line to be added. Both fixes add method calls of `ImageList.put(int, Image)` when the if condition `index == -1` is not satisfied. If we can learn and apply such recurring fix patterns automatically, we can save developers time by showing them the correct fix.

```
1  public void setImage (Image image) {
2    checkWidget();
3    if ((style & SWT.SEPARATOR) != 0) return;
4    super.setImage (image);
5    if (!OS.GTK_IS_IMAGE_MENU_ITEM (handle)) return;
6    if (image != null) { //context
7      ImageList imageList = parent.imageList;
8      if (imageList == null)
9        imageList = parent.imageList = new ImageList();
10     int imageIndex = imageList.indexOf (image);
11-    if (imageIndex == -1)
12-      imageIndex = imageList.add (image);
13+    if (imageIndex == -1) {
14+      imageIndex = imageList.add (image);
15+    } else {
16+      imageList.put (imageIndex, image);
17+    }
18     ...
19   } else {
20     OS.gtk_image_menu_item_set_image (handle, 0);
21   }
22 }
```

Figure 4.1: *APARE* generates this fix automatically, identical to developers' fix, for Eclipse SWT bug 94003.

```
1  int imageIndex (Image image) {
2    if (image == null) //context
3      return OS.I_IMAGENONE;
4    if (imageList == null) { //context
5      Rectangle bounds = image.getBounds();
6      imageList = display.getImageList(
7           new Point (bounds.width, bounds.height));
8-   int index = imageList.indexOf (image);
9-   if (index == -1)
10-    index = imageList.add (image);
11       ...
12   }
13   int index = imageList.indexOf (image);
14+ if (index == -1) {
15+      index = imageList.add (image);
16+  } else {
17+      imageList.put (index, image);
18+ }
19 ...
20 }
```

Figure 4.2: *APARE* leverages this fix to generate the fix in Figure 4.1.

Project-specific recurring fixes are not well handled by current general automated program repair techniques because such fixes are relatively large in size (or are embedded in complex fixes) and often contain rich project-specific semantics (e.g., adding an invocation

of a project-specific method). For example, the complete fix (i.e., making all failing test cases pass) for SWT bug 94003 contains 154 lines in seven methods (61 deleted lines and 93 added lines). In this work, we propose *APARE* to complement existing repair techniques in generating project-specific recurring fixes.

Current general automated program repair techniques search for a correct repair in a constructed search space. There are different ways of constructing the search space: (1) applying modification operators (add, delete, and mutate) on the code (GenProg [70], RSRepair [116] and Kali [117]), (2) using pre-defined or automatically-learnt common fix patterns to create a repair (PAR [61], SPR [83], and Genesis [82]), and/or (3) using constraint solving to fix defective conditions (SPR [83], Angelix [94], SemFix [107], Qlose [31]). Type (1) techniques do not work well in generating recurring fixes because recurring fixes are often complex in size and result in large search spaces that make type (1) techniques intractable. In fact, many fixes of type (1) techniques are single line deletions to avoid buggy behaviours [117]. Type (2) techniques leverage common fix patterns across projects but do not recognize project-specific fix patterns. For example, Genesis [82] concludes common fix patterns for specific types of bugs, such as null pointer exceptions, and class cast exceptions. Type (3) techniques target the bugs that can be fixed by repairing the defective conditions, thus they may not be effective in generating recurring fixes with rich project-specific semantics.

Recent advances in automated program repair leverage past fixes to rank the patch candidates in the search space. For example, Prophet [85] ranks the patch candidates in SPR's search space based on a probability model that is trained from past fixes. Le et al. [68] propose (referred as *HistoryDriven*) to rank and select the top patch candidates based on the code similarity between the patch candidates and mined frequent cross-project fix patterns. These techniques may not generate project-specific recurring fixes since the mined fix patterns or the built probability model is based on cross-project fix patterns, which do not cover project-specific semantics.

Recurring fixes may be full or partial to complete fixes (i.e., making all failing test cases pass). When partial, recurring fixes are still valuable and can help developers write complete fixes. Presenting partial fixes to developers can significantly improve debugging correctness [58, 135]. However, recurring fixes that are partial do not mean such fixes are recurring at the statement level (e.g., deleting a method invocation). Particularly in this work, we target generating recurring fixes that are *complete at the method level*, but maybe partial to the complete fix. In this work, a complete fix to one method is referred to as a *method fix*. As an example, the fix in Fig. 4.1 is a method fix generated by *APARE*. While the fix alone does not completely fix the bug, it is a correct and a complete fix for the faulty method `setImage` (The complete fix contains seven method fixes). *APARE* generates four

correct method fixes for this bug and the remaining three are not recurring fixes.

4.1.1 Automatically learning and applying project-specific fix patterns: state of the art and challenges

The central issue to generating project-specific recurring fixes is in learning and applying project-specific fix patterns. The techniques in the systematic editing tools Sydit [95] and LASE [96] may be used to automatically learn fix patterns at the method level. In fact, with a few modifications that allow them to search through a project's commit history, they can theoretically be used to build a database of project-specific fix patterns and use those fix patterns to automatically generate recurring fixes.

To determine whether these tools can handle automated repair tasks on project-specific recurring bugs, this work conducts an empirical study (§4.3) on recurring fixes. Sydit and LASE are unsuitable for automated repair since they produce an unacceptably large number of false positives and false negatives. I identified the following challenges that prevent Sydit and LASE from working well for generating recurring fixes.

(i) More relaxed and accurate context matching: Sydit requires that the ***contexts***—data and control dependencies, and *changed* lines (i.e., lines that start with -) of the fix—to be **identical** at the Abstract Syntax Tree (AST) level.

The empirical study finds that *27–92% of recurring method fixes have different contexts* at the AST level (§4.3.4). For example, the contexts are different in Fig. 4.1 and 4.2. According to Sydit's definition of context, line 2 and 4 are contexts of the edits in Fig. 4.2 and line 6 is a context of the edits in Fig. 4.1, as annotated by comments. Therefore, Sydit cannot find a matched context for line 4 in Fig. 4.2. Relaxing some context requirements is needed to match the two examples. In addition, Sydit cannot match line 2 in Fig. 4.2 with line 6 in Fig. 4.1, which are semantically equivalent (`if (image == NULL) return; foo();` is equivalent to `if (image != NULL) foo();`), because they are syntactically different. A new solution is needed to match the *semantically equivalent but syntactically different* contexts.

(ii) More precise selection of past fixes and faulty locations: Since Sydit and LASE are designed for refactoring code instead of fixing bugs, they require developers to provide information such as the faulty methods and past fixes that are similar to the given bug. Developers often do not know such information. Simply using faulty methods identified by fault localization and all past fixes are imprecise. The evaluation in Section 4.5.1 shows

[1]Sydit accepts minor context differences, e.g., `if(ptr)` and `if(foo())` can be matched.

that directly applying Sydit and LASE to 20 recurring bugs achieves a precision of 0.003% and 0.08% only. *Precision* is the portion of generated fixes that are correct. A new solution is needed to precisely select fix patterns and generate correct fixes for faulty locations.

In summary, while the systematic editing tools Sydit and LASE can be applied to automated repair, the high number of false positives and false negatives make it impractical. A program repair approach that automatically learns fix patterns should significantly reduce the manual effort and increase the effectiveness of program repair. However, these techniques are promising and can be made practical by solving the challenges identified in our empirical study.

4.2 The Main Contributions of this Chapter

This chapter proposes APARE—a novel automated repair technique that learns fix patterns automatically to effectively generate recurring fixes.

Given a bug with relevant test cases, *APARE* automatically learns patterns at the Abstract Syntax Tree (AST) level from past fixes. Next, *APARE* uses fault localization techniques [8] to identify likely faulty methods, identifies applicable fix patterns based on contexts, and generates recurring fixes from the fix patterns automatically.

To address the challenges above (Section 3.1.1), this work implements four main techniques: (1) *AST-based program analysis* to learn fix patterns, (2) *fix pattern filtering* to automatically select past fixes that are big enough for fix pattern matching, (3) a new *matching algorithm* named **semantics-aware percentage context matching algorithm** to match the similar contexts (both syntactic and semantic similarities) between past fixes and faulty locations with high accuracy, (4) *generating new fixes* by synthesizing identifiers and *checking* the validity of the generated fixes in new locations. Techniques (2)–(4) are novel, while (1) is similar to that of Sydit and LASE.

These techniques allow *APARE* to remove the manual effort in learning fix patterns, which is essential to generate recurring fixes and is required by previous work such as PAR and R2Fix.

In general, techniques (2)–(4) enable *APARE* to identify more accurate matches between past fixes and faulty locations to generate correct recurring fixes. *APARE* can generate fixes that are similar but not identical to past fixes; APARE *can generate fixes even if the contexts are different.*

Table 4.1: Studied Software

Software	LOC	Bugs	Commits of Bug Fixing	Method Fixes
Eclipse SWT	731K	1,218	1,752	6,066
Eclipse JDT	2.29M	1,694	4,914	15,374
ZK Web Framework	778K	563	1,064	873
OpenJPA	505K	778	1,345	3,358
Wicket	319K	1,789	2,463	5,420

This chapter evaluates the correctness and completeness of *APARE* on recurring bugs from five projects. On the 20 randomly selected recurring bugs, *APARE* generates 20 correct method fixes that the search-based repair technique RSRepair and the pattern based technique PAR, cannot generate. The precision of *APARE* is acceptable compared to that of existing automated repair techniques (6–23%) [83, 70].

For the 20 recurring bugs, *APARE* learns 3,838 fix patterns automatically and uses them to generate 34 method fixes, 24 of which (70.6%) are correct; 83.3% (20 out of 24) are identical to developer fixes; 5 are complete fixes; 2 are over 75% complete fixes compared to the complete fixes by developers. *APARE* is robust—on 5 randomly selected non-recurring bugs, *APARE* only generated incorrect method fixes for one of the bugs.

This chapter makes the following contributions:

- a study towards understanding project-specific recurring fixes;
- a novel automated technique (*APARE*) that targets project-specific recurring fixes;
- an evaluation of *APARE* on 25 bugs (20 recurring and 5 non-recurring) in real-world Java projects;
- a comparison of *APARE* against general automated repair tools (RSRepair, PAR, and HistoryDriven) on the collected 20 recurring bugs;

4.3 A Study of Project-Specific Recurring Fixes

The main *goal* of the study is to obtain insights on how to learn and apply fix patterns for automatically generating recurring fixes. Inspired by Sydit, we study the *contexts* (i.e., control and data flows) of recurring fixes.

Totally five open source systems are studied (Table 4.1), which have been commonly used by previous work about recurring fixes and systematic edits [95, 96, 108]. The following bug fixing time periods are studied: from 2004/07 to 2006/07 for SWT and JDT, from

2006/05 to 2014/07 for ZK, from 2006/05 to 2014/12 for OpenJPA and 2004/09 to 2014/12 for Wicket.

4.3.1 Identifying Bug Fixing Commits

Bug-fixing commits are identified automatically by scanning the commit logs for bug IDs defined in bug databases. This automated process uses the same heuristics that are proposed by Mockus and Votta [99] and used by previous work [114], which is to look for bug IDs in commit logs by identifying numbers that are separated by delimiters including "#().,\n\t\r\f". Table 4.1 presents the number of bug fixing commits and the number of method fixes in the studied software.

4.3.2 Identifying Candidate Recurring Fixes

This empirical study focuses on recurring method fixes, which are two method fixes that are identical or similar to each other, also referred to as *recurring method fix pairs*. To conduct the study, a large sample of recurring method fix pairs should be identified first. The percentage of recurring method fix pairs is low. For example, if a project has 1,000 method fixes and 20% of them are recurring fixes (fixes that are identical or similar to another fix), the percentage of recurring fix *pairs* would be at most 4% and can be as low as 0.02%[1]. Therefore, if random 100 pairs are sampled, 0–4 pairs would be expected to be collected, which would be too small for a comprehensive study. Since recurring fixes must be similar to some extent, pairs of candidate recurring fixes are identified if that are at least 60% similar at the Abstract Syntax Tree (AST) level (details at the end of §4.3.2) to filter out pairs that are unlikely to be similar. Although this approach might miss some recurring fix pairs, this is the best effort since random sampling is prohibitively expensive. The threshold is set to be relatively low, i.e., 60%, to minimize missing pairs of recurring fix.

I propose and apply the following AST-based technique to identify potential recurring fixes automatically.

Representing Fixes At the AST Level. A widely used tool, ChangeDistiller [35] is chosen to represent fixes as syntactical instead of textual differences (*diff*). Specifically, for

[1]If all 20% of the recurring fixes (200 in total) are similar to each other, then the percentage of recurring fix pairs is $\binom{200}{2}/\binom{1000}{2}$=4%. If 200 of the recurring fixes form 100 recurring pairs where each pair is similar to each other but different from the rest, then the percentage of recurring fix pairs is $100/\binom{1000}{2}$=0.02%.

```
1 - OS.gtk_entry_set_max_length (entryHandle, limit);
2 + if (entryHandle!=0)
3 +   OS.gtk_entry_set_max_length (entryHandle,limit);
```

Figure 4.3: A Bug Fix in Diff Format.

```
1 INSERT: if (entryHandle!=0)
2 PARENT_CHANGE_OF:
3   OS.gtk_entry_set_max_length(entryHandle, limit);
```

Figure 4.4: Two Edit Operations from ChangeDistiller for the Fix in Figure 4.3

two versions `A` and `B` of a program file (the bug fix is the changes from version `A` to version `B`), ChangeDistiller calculates the minimum *tree edits* needed from the AST of version `A` to the AST of version `B`. Figure 4.4 shows ChangeDistiller's representation of the fix in Figure 4.3. It first inserts an if statement, and then changes the parent of the method invocation to the newly inserted if statement.

Associating Relevant Edits. Two edits are associated as relevant if they are either *defs* or *uses* of same variables. For example, in Figure 4.4, the two edits are associated because they are *uses* of `entryHandle`.

Similarity Comparison of Bug Fixes. Given two sequences of edits of two bug fixes, the Longest Common Subsequence (LCS) algorithm is used to find matching edits between the two sequences, and calculate the similarity using:

$$SimilarityMetric = \frac{2 \times \text{No. of Matched Edits in Two Sequences}}{\text{Sum of the No. of Edits in the Two Sequences}} \tag{4.1}$$

Only pairs of fixes whose $SimilarityMetric$ is 60% or higher are kept as candidate pairs. Also, a minimal number of matched edits of three is required because if two bug fixes have only two or fewer edits in common, they are unlikely similar. Two *edits are matched* if and only if they have the same change type (e.g., insert), statement type (e.g., if statement), and *similar* relevant edits. Relevant edits of two edits are *similar* if the number of matched edits between the two is above a threshold (50% in our experiment). In this case, edits match if they have the same change type and statement type, but similar relevant edits are not required to match.

4.3.3 Manual Examination of Recurring Fixes

Manual analysis is performed on a *random* sample of 150 pairs of candidate recurring fixes from each project (a total of 750 pairs for the five projects). One of my co-authors and I (see "Statement of Contributions"), examine each pair of bug fixes independently and then reach an agreement on the results, i.e., whether two fixes are similar, and whether the contexts of recurring pairs are identical or different. If we disagree on a pair, another co-author examines the pair, and we take the majority voting.

Specifically, we manually check whether a pair of bug fixes are identical, similar, or different. Identical and similar fix pairs are recurring fixes. A fix pair is similar if they satisfy one or multiple of the following conditions: (i) fix A is a subset of fix B or vice versa, (ii) source code has the same method names with different types or different number of arguments, (iii) fix A and fix B are identical except branch conditions' syntax is slightly different, and (iv) source code statements are semantically equivalent (§ 4.4.3-(1)). All other fix pairs are considered different fixes.

We check whether recurring fixes have identical or different contexts. Inspired by Sydit [95] and LASE [96], we define *context* as changed lines (i.e., lines that start with -) of a fix and their backward control and data dependencies. Two recurring fixes have identical context if their control and data dependencies are identical. Otherwise, we consider the two fixes have different contexts. Fig. 4.1 and 4.2 are recurring fixes with different contexts. The contexts are different because the control dependencies in Fig. 4.1 (line 6) and those in Fig. 4.2 (line 2 and 4) are different.

4.3.4 Study Results

Table 4.2 shows the results of manual examination of recurring fixes. In total, we verified 242 true recurring fixes (54 from SWT, 63 from JDT, 38 from ZK, 61 from OpenJPA and 26 from Wicket).

Manual verification shows that **27–92% of the recurring fixes have different contexts**. *The results of our empirical study show that state-of-the-art approaches [95] would not work for most of the recurring fixes due to the different contexts. This empirical finding exposes challenges of learning and applying recurring fix patterns (§4.1.1). New techniques are needed to generate recurring fixes with different contexts.*

Table 4.2: Summary of Recurring Fixes

Software	Candidate Pair	Sample Size	Recurring Fixes	Identifical Context	Different Context
Eclipse SWT	5,439	150	54	37%	63%
Eclipse JDT	19,410	150	63	38%	62%
ZK Web Framework	404	150	38	8%	92%
OpenJPA	1,991	150	61	49%	51%
Wicket	4,876	150	26	73%	27%

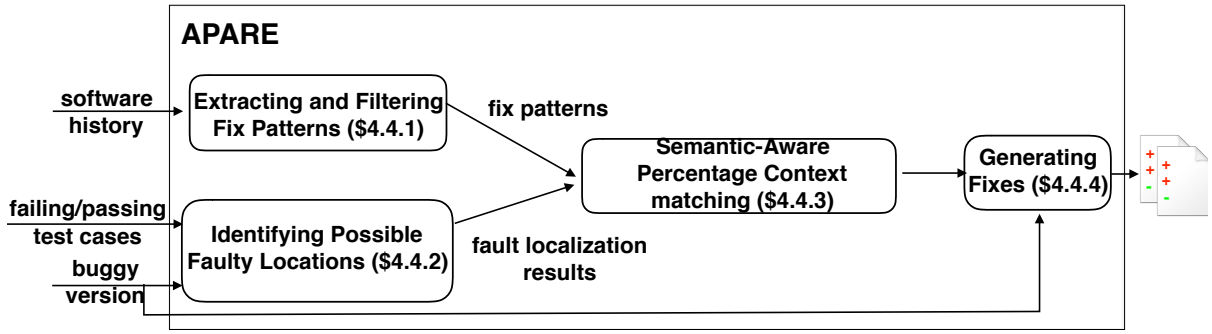

Figure 4.5: The overview of *APARE*

4.4 Design of *APARE*

This section describes the four components of *APARE* (as illustrated in Figure 4.5). *APARE* generates fix patterns from all past bug fixes that were committed before this bug was fixed (§ 4.4.1). Given a set of test cases including at least one failing test case, it then uses fault localization techniques to identify possible faulty locations (§ 4.4.2), and performs context matching to identify fix locations and past fix patterns that may be applicable to the given bug (§ 4.4.3). Finally *APARE* applies fix patterns to applicable fix locations (§ 4.4.4).

As discussed in §4.3, 27–92% recurring fixes have different contexts, thus techniques such as Sydit that require identical context matching cannot be applied directly. *Simply relaxing the identical context matching requirement would cause many incorrect fixes generated at irrelevant locations* [95, 96]. *APARE* addresses this open challenge by using a *semantics-aware percentage context matching* algorithm. Inspired by the empirical study results, the matching algorithm ignores irrelevant contexts, and uses semantic matching to match semantically equivalent but syntactically different contexts.

Manually identifying similar fixes in history is time-consuming. Given the potentially faulty methods identified by fault localization techniques, a simple heuristic that developers can use is to search for all past fixes that modify these faulty methods and other methods with identical names to these faulty methods. However, this simple heuristic could return many past fixes and the burden would be on the developers to manually select one or many recurring past method fixes from them. For example, to fix bug 139329 in Eclipse SWT, the above heuristic returns 589 past method fixes for developers to manually examine.

The design of *APARE* favours high precision—ensuring generated fixes are correct, instead of favoring high recall, i.e., generating as many correct fixes as possible. The reason is two fold. First, incorrect and low quality fixes can hurt debugging correctness [135]. Second, even if no fixes are generated for a bug, developers can still follow their normal process to fix the bug, in which case *APARE* adds no cost to the process. Since automatic fix generation is extremely challenging, it is still valuable even if the recall is low (PAR, GenProg and SPR [83] fix only 1–17% of bugs automatically [61]).

4.4.1 Extracting and Filtering Fix Patterns

APARE learns general and accurate fix patterns at the AST level from past fixes. Bug fixing commits are identified using links to bug reports in commit messages, as illustrated in Section 4.3. Fix patterns include edits at the AST level that transform the code and contexts of the edits. Contexts are included since they are crucial in deciding if the edits can be applied to new locations (§ 4.3). For each fix pattern, *APARE* abstracts identifiers (e.g., variable names) to make it general to other fix locations. The techniques described in (1) and (2) below are similar to those in Sydit [95], so the these techniques are briefly summarized.

(1) Extracting Edits at the AST Level. *APARE* uses ChangeDistiller [35] to generate program differences at the AST level. *ChangeDistiller* uses four types of edit operations to represent program differences: *update*, *move*, *delete*, and *insert*.

update(u, v): Statement u is updated to v.

move(u, v, k): Statement u is moved from its current parent to v as a child at index k.

delete(u): Statement u is deleted.

insert(u, v, k): Statement u is inserted as the child of the statement v at index k.

Fig. 4.2 shows a bug fix in diff format. The corresponding edits on an AST are as follows. For each edit, we show the relevant line numbers. For example, the first edit "*move ('if*

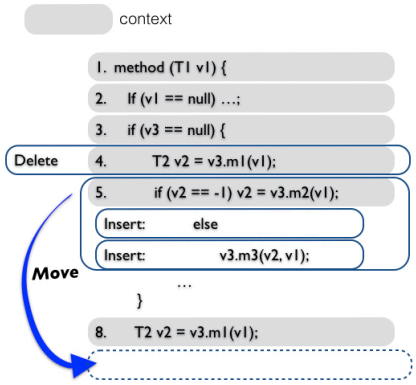

Figure 4.6: The complete fix pattern extracted by *APARE* from the motivation example (Figure 4.2). The grayed statements are the context of the fix – data and control dependencies. All identifiers are abstracted and ready for matching against new locations.

(index = -1)', 'imageIndex', 3) @L9, L14" means to move the `if` statement that is originally at line 9 to be the child of the method `imageIndex` at index 3, and the new location is line 14.

1. move ('if (index = -1)', 'imageIndex', 3) @L9, L14
2. insert (else_statement, if_statement='if (index == -1)', 1) @L16
3. insert ('imageList.put (index, image);', else_statement, 0) @L17
4. delete @L8

(2) Extracting Data and Control Dependencies. *APARE* extracts control and data dependencies of each statement that is inserted, deleted, updated and moved. Specifically, for two versions of code (*original* and *patched*), *APARE* computes two backward program slices (one for control dependencies and one for data dependencies) for each changed statement, where the statements are the seeds for the slice. Then, *APARE* locates the corresponding nodes of the control and data dependencies of the *patched* version in the *original* code, and takes a union with the dependencies of the *original* version to make a complete dependency set. Taking a union with the dependencies in the *patched* version is crucial because applicable edits should follow context in the *patched* version as well.

(3) Filtering Fix Patterns. Many learnt fix patterns have only one or two lines of change, e.g., 'adding a null check'. Such small fix patterns are often applicable to many locations

and thus cause many incorrect fixes to be generated. Since *APARE* aims to generate fixes with high precision, *APARE* filters out these small fix patterns, and focuses on generating more complex fixes. In addition, such small patterns have been successfully applied by previous work such as PAR [61] and R2Fix [75], and are not the focus of *APARE*. Instead, *APARE* aims to complement existing techniques. Specifically, *APARE* keeps only the fix patterns for which both the number of edits and the number of contexts are greater than two.

An example of the fix pattern extracted by *APARE* Figure 4.6 shows the fix pattern that *APARE* extracts from the past fix in Figure 4.2. The fix pattern is composed of four edits that are described in (1) and the context. The grayed statements are the context related to the fix based on control and data dependencies. As described in (1), all the identifiers of the fix are abstracted in the fix pattern, e.g., `image` is abstracted to `v1`.

4.4.2 Identifying Possible Faulty Locations

APARE uses fault localization techniques [8] to narrow down the scope of applicable methods by identifying possibly faulty locations. *APARE* collects execution information from both failing and passing test cases, and ranks the executed lines using the *Jaccard* similarity coefficient. *APARE* selects top ranked lines and uses their corresponding methods as potentially faulty methods. In this experiment, *APARE* uses either top 5% or top 200 ranked lines when selecting top 5% ranked lines yields more than 200 lines; this significantly limits the size of potentially faulty methods when repairing programs with long execution paths. I observed that methods in the same class involve recurring fixes; however test cases in the bug reports only trigger one method out of several faulty methods in the same class. Therefore, *APARE* includes all methods from the classes of the identified methods, but only from relatively small classes which have fewer than 70 methods. For classes with more than 70 methods, only the methods that contain the top ranked lines are included as faulty methods.

4.4.3 Semantics-Aware Percentage Context Matching to Find Applicable Fix Patterns

Guided by the empirical study of recurring fixes, *APARE* proposes a new matching algorithm which considers both *semantic* and *syntactic* similarities, and tolerates some differences in context: ***semantics-aware percentage context matching*** algorithm (Algorithm 1). Compared to Sydit's matching algorithm, *APARE* proposes three key techniques to enable

Algorithm 1: Semantics-Aware Percentage Context Matching Algorithm
Input: FP, Method /* the fix pattern and the target method */
Output: m_id /* a set of matched identifiers from FP to Method */
Output: matched /* whether FP and Method are matched */
1 m_leaves ← ∅; /* a set of matched leaf nodes */
2 m_nodes ← ∅; /* a set of matched all nodes*/
3 matched ← false; /*whether FP and Method are matched*/
4 **ETranform**(FP); **ETranform**(Method);
5 **matchLeafNodesSemantically**(FP, Method, m_leaves);
6 m_id ← **matchInnerNodesSemantically**(FP, Method, m_leaves, m_nodes);
7 matched ← **isPercentageMatched**(FP, m_nodes);

more accurate matching: (1) *equivalence transformations* to identify more matches of similar contexts precisely, (2) *semantic matching* in addition to syntactic matching for better accuracy, and (3) *percentage context matching* to match similar but not identical context.

Algorithm 1 takes a fix pattern and a target method in AST format as input and finds matched AST nodes and identifiers between them. First, it performs (1) *equivalence transformations* (line 4) on both the fix pattern and the target method to transform semantically-equivalent but syntactically-different code snippets into syntactically-identical code snippets. Second, it finds semantically- and syntactically-similar pairs of leaf nodes between the two ASTs (line 5). Third, for each pair of matched leaf nodes, it matches the inner nodes (non-leaf nodes) on the two paths from the roots of the ASTs to the two leaf nodes (line 6) by tolerating syntactic differences caused by unmatched control dependencies. Both the second and third steps leverage (2) *semantic matching* to ensure the matched nodes are semantically similar. Last, the score of similarity is calculated by Formula 4.2 based on the matched nodes. A fix pattern and a target method are matched if the score passes a threshold. The third and last steps perform (3) *percentage context matching*.

This section explains the three key techniques. Also, it presents explanations of each function used in Algorithm 1 when (3) *percentage context matching* is explained.

(1) Equivalence Transformations. As discovered in our empirical study, recurring fixes can have different but semantically equivalent contexts. To improve the accuracy of AST-level context matching between fix patterns and target methods, the following equivalence pairs are defined, which are used by *APARE* to transform both fix patterns and target methods. Fig. 4.7 shows an example of two recurring fixes from SWT that are equivalence under E6.

```
- double lineWidth = Cairo.cairo_get_line_width(handle);
- Cairo.cairo_set_line_width(handle,
lineWidth/(width/2f));
```

(a)

```
- Cairo.cairo_set_line_width(cairo,
Cairo.cairo_get_line_width(cairo)/(width/2f));
```

(b)

Figure 4.7: An Example of Equivalence 6 (E6)

E1: if (!v) then **ThenBlock** else **ElseBlock**
≡ if (v) then **ElseBlock** else **ThenBlock**

E2: if (v != Expression) then **ThenBlock** else **ElseBlock**
≡ if (v == Expression) then **ElseBlock** else **ThenBlock**

E3: if (v) then **return**; **OtherStatements**
≡ if (v) then **return**; else **OtherStatements**

E4: if (v) then ... else **return**; **OtherStatements**
≡ if (v) then ... **OtherStatements** else **return**;

E5: Object v; v = Expression;
≡ Object v = Expression;

E6: (Object) v = Expression;
M(..., v, ...); (no `defs` of `v` between)
≡ M(..., Expression, ...);

Equivalence transformations are crucial, because they improve the accuracy of AST context matching by increasing the syntactic similarity of AST structures. For example, line 6 in Fig. 4.1 and line 2 in Fig. 4.2 are semantically equivalent (**E2** and **E3**), yet they have different AST structures, which prevents them and their descendent AST nodes from being matched.

After equivalence transformations on fix patterns and target methods, *APARE* updates the edits accordingly. For example, using E3, "*1. move ('if (index = -1)', 'imageIndex', 3)*" becomes "*1. move ('if (index = -1)', 'else_statement', 3)*". '*else_statement*' is the `else` statement introduced by the **E3** transformation.

(2) Semantic Matching on Method Invocations. As a refactoring tool, Sydit does not require method, variable, or type names to match. It can afford to do so because it asks

developers to pick a past fix and select a target method to apply to. As a fully automatic tool, *APARE* needs to identify applicable fixes from many past fixes, and applies them to many methods because developers often do not know what the faulty methods are. Such wide application causes many inaccuracies [95, 96] (supported by §4.5.1). To address this issue, *APARE*'s AST matching algorithm is restricted by matching only method invocations with semantic similarity, because we find that recurring fixes often involve changes to method invocations with semantic similarity. Recall that favoring high precision is our design choice. The following example shows how *APARE* decides whether two method invocations are semantically similar.

For example, in order to compare method invocations `OS.XQuery Color(display, data.colormap, color)` and `OS.gdk_colormap_query_color(colormap, color.pixel, color)`, a vector of tokens for each method convocation is constructed based on naming conventions and delimiters. The first vector is <**OS**, X, **Query**, **color**, display, data, **colormap**, **color**>, and the second vector is <**OS**, gdk, **colormap**, **query**, **color**, pixel, **color**>. The *matched* names are in bold. These two vectors are compared and the percentage of *matched* words in two vectors is calculated, which is 10/15. Two method invocations are semantically similar if the percentage of *matched* words passes a threshold ($T_{semantic}$). This threshold is set to be *0.6* in the evaluation.

(3) Percentage Context Matching. The matching algorithm (Algorithm 1) enables **percentage matching** using two steps. First, `matchInnerNodesSemantically` (line 6 in Algorithm 1) tolerates syntactic differences of AST structures caused by unmatched control dependencies. The details are in Algorithm 2. Second, after collecting matches of AST nodes, *APARE* leverages the score of similarity (Formula 4.2) to tolerate unmatched contexts (control/data dependencies and changed code). Technique (2) *semantic matching* is leveraged in both leaf and inner node matching to improve the matching accuracy since semantic similarity is required.

Algorithm 2 allows some inner nodes to be ignored from both the fix pattern and the target method (lines 11–15). For example, the leaf node "`index = imageList.add(image);`" (line 10 in Fig. 4.2) and the leaf node in line 12 of Fig. 4.1 match exactly. However, the two paths from the roots to the leaf nodes are not matched because the inner nodes on the paths fail to match after equivalence transformations (e.g., lines 3 and 5 of Fig. 4.1 have no matching nodes in Fig. 4.2). Then these unmatched inner nodes are added to the sets of ignored nodes (`ignoredFP` and `ignoredM`) from the fix pattern and the target method respectively, so that they are always ignored (line 18). Note that the sets of ignored nodes are assigned only in the first iteration (lines 5–22). During the rest of the iterations, they

Algorithm 2: matchInnerNodesSemantically(FixPattern fp, Method method, LeafNodeMatches m_leaves, NodeMatches m_nodes)

```
1  ignored ← false;
2  m_id ← ∅; /*a set of identifier matches*/
3  ignoredFP ← ∅; /* a set of ignored nodes from the fix pattern*/
4  ignoredM ← ∅; /* a set of ignored nodes from the method*/
5  repeat
6      foreach unique (x, y) ∈ m_leaves do
7          /* unique means x matches with y only in m_leaves*/
8          if isPathMatch(x, y) then
9              addNodesFromPaths(x, y, m_nodes);
10         else
11             if !ignored then
12                 /* get paths from root to leaf node */
13                 p1 = root_FP ⇝ x; p2 = root_method ⇝ y;
14                 addNodesFromPathTolerance(p1, p2, m_nodes, ignoredFP,
                   ignoredM);
15             end
16         end
17     end
18     ignored ← true; /*only ignore nodes at the first iteration*/
19     addNodeMappingsFromPath(m_nodes, m_leaves, ignoredFP, ignoredM);
20     buildIdentifierMappings(m_id, m_nodes);
21     m_leaves ← m_leaves ∪ relaxConstraints(fp, m_nodes);
22 until m_leaves does not change;
23 return m_id
```

are not added, as `ignored` is set to `true` in line 18. Only in the first iteration, the matching requirement is relaxed because the mappings of leaf nodes are the most strict, which require *exact* matching of AST labels while the rest of the iteration require only *type* matching in `relaxConstraints` (line 21).

The following list of methods presents the functionalities of methods used in Algorithm 1 and 2 below. Compared to Sydit, the following three methods are new. Time and space complexity is provided for each method, as well as for Algorithm 1 and Algorithm 2 (see the last of the following list).

- **ETransform(FP)**, **ETransform(Method)**: Performs equivalence transformations on the

fix pattern and the target method.
Time complexity: $O(n)$, where n is the number of statements in `FP` or `Method`.
Space complexity: constant spaces.

- **matchLeafNodesSemantically(fp, method, m_leaves)**: For every pair of leaf nodes (x, y) from `fp` and `method`, if (i) the AST labels of x and y are equivalent (e.g., "T1 v1 = M(v3)" and "T2 v2 = M(v4)"), and (ii) x and y are semantically matched (refer to *(2) Semantic Matching on Method Invocations* as described earlier), `(x, y)` is added to `m_leaves`.
  Time complexity: $O(n * m)$, where n is the number of leaf nodes in `fp` and m is the number of leaf nodes in `method`.
  Space complexity: $n + m$, where n is the number of leaf nodes in `fp` and m is the number of leaf nodes in `method`.
- **isPercentageMatched (FP, m_nodes**): Based on matches of AST nodes (`m_nodes`), the score of similarity is calculated as Formula 4.2. It returns true if the score passes the threshold, false otherwise.
  Time complexity: $O(n + m)$, where n is the number of AST nodes in `FP`, and m is the number of matches of AST nodes in `m_nodes`.
  Space complexity: constant space.

The first three below are modified compared to Sydit. The rest are similar to those of Sydit.

- **addNodesFromPathTolerance(p1, p2, m_nodes, ignoredFP, ignoredM)**: For every node **x** in p1, it finds a node **y** in p2 that is exactly matched with **x** (x and y have equivalent AST labels), and adds (x, y) to `m_nodes`. If **y** cannot be found, it adds **x** to `ignoredFP`. Finally, it adds all unmatched nodes in p2 to `ignoredM`.
  Time complexity: $O(n)$, where n is the length of the longer path of `p1` and `p2`.
  Space complexity: $n + m$, where n is the length of `p1` and m is the length of `p2`.
- **addNodeMappingsFromPath(m_nodes, m_leaves, ignoredFP, ignoredM)**: From multiple leaf node mappings, it adds inner node mappings by selecting the best leaf node mapping. It uses *pathMatchScore* and *LCSMatchScore* from Sydit [95]. Comparing to Sydit, the modification is that `ignoredFP` and `ignoredM` are excluded from the paths from the root to the leaf node.
  Time complexity: $O(n)$, where n is the number of node matches in `m_nodes`.
  Space complexity: constant spaces.
- **relaxConstraints(fp, m_nodes)**: For unmatched nodes in `fp`, it finds type matched and semantically matched nodes by leveraging parent-child relationship and the mappings in m_nodes.

Time complexity: $O(n * m)$, where n is the number of unmatched nodes in `fp`, and m is the number of node matches in `m_nodes`.
Space complexity: constant spaces.

- `isPathMatch(x, y, m_nodes)`: Compares the paths (p1, p2) from root$\rightsquigarrow$x and root$\rightsquigarrow$y, and returns **true** if for each node in p1, there is a node in p2 which is type matched with this node and they are in the same position.
  Time complexity: $O(n)$, where n is the length of the longer path of `p1` and `p2`.
  Space complexity: constant spaces.
- `addNodesFromPath(p1, p2, m_nodes)`: Adds node pair (u, v) to `m_nodes`, if u and v are type matched and on the same position of `p1` and `p2`.
  Time complexity: $O(n)$, where n is the length of the longer path of of `p1` and `p2`.
  Space complexity: n, where n is the length of the longer path of `p1` and `p2`.
- `buildIdentifierMappings(m_id, m_nodes)`: Adds identifier mappings to `m_id` based on the node mappings in `m_nodes` and removes conflicting mappings.
  Time complexity: $O(n)$, where n is the number of node mappings in `m_nodes`.
  Space complexity: n, where n is the number of node mappings in `m_nodes`.

The time complexity of Algorithm 1 and Algorithm 2 is $O(n + m)$, and the space complexity is $z * (n + m)$: n is the number of AST nodes in the fix pattern (FP), m is the number of AST nodes in the method to be matched ($Method$, see the input of Algorithm 1 and the method declaration in Algorithm 2), and z is a value that depends on the size of identifier mapppings established (in line 20 of Algorithm 2).

In addition to Algorithm 2, *APARE* uses the score of similarity defined in Formula 4.2 to tolerate unmatched contexts (line 7 in Algorithm 1). To balance different types of context (control/data dependencies and changed code), *APARE* gives each type a *total* weight, named W_{ctrl}, W_{data}, and W_{code}. The *total* weight for each type is distributed to each context based on the portion of edits that depend on this context. Then *each* control dependency statement receives different weight as W_{Ectrl}=$W_{ctrl} * Perc_{edit}$, where $Perc_{edit}$ is the portion of total edits that depend on this context. The weight of each data dependency (W_{Edata}) and each changed code (W_{Ecode}) is calculated in a similar fashion. The *Score* of matched context is calculated as follows,

$$Score = \frac{\sum_{MatchedCtrl} W_{Ectrl} + \sum_{MatchedData} W_{Edata} + \sum_{MatchedCode} W_{Ecode}}{W_{ctrl} + W_{data} + W_{code}} \tag{4.2}$$

Our evaluation uses the threshold of *Score*, named $T_{percentage}$, as 0.7. W_{ctrl} is 2, W_{data} is 2 and W_{code} is 1.

4.4.4 Generating Fixes for Fix Locations

APARE generates fixes for new locations from fix patterns if they are percentage-matched (§4.4.3). An edit is applicable only if essential nodes have matched nodes in the faulty method. For example, for `insert` edits, the parent nodes must exist in the faulty method.

Moreover, *APARE* extends Sydit's fix generation to significantly reduce incorrect fixes. *APARE* leverages the following techniques to 1) synthesize identifiers and method invocations; and 2) filter out invalid edits. First, *APARE* replaces abstract identifiers with concrete identifiers based on the identifier mappings established by Algorithm 1 and synthesizes identifiers if needed. Specifically, if there is an identifier that does not have a mapping in the fix location, *APARE* checks 1) if this is a newly inserted variable by a previous applicable edit, and 2) if there is a variable with the same name in the scope. If neither is satisfied, the edit is not applicable. *APARE* replaces method invocations in a similar fashion. If a method name from a past fix is not mapped with one method name in fix location, *APARE* tries to synthesize a proper method invocation based on 1) methods with identical names in the scope; and 2) types of parameters.

Second, *APARE* conducts *sanity* checks to ensure that generated fixes do not break data or control dependencies in the patched version. For example, deleting or moving a variable declaration is applicable only when the variable is not used by other statements. *APARE* conducts *necessity* checks after the sanity checks. For an edit that is *insert*, or *move*, if the inserted/moved node is a variable declaration, this edit is applicable only if the declared variable is used and does not conflict with existing variable declarations.

4.5 Evaluation

This section describes the methodology for evaluating *APARE* and how to conduct the comparison with current general automated program repair (e.g., RSRepair, PAR and HistoryDriven). The following four research questions are answered in this evaluation:

RQ1 Complementarity: *Can* APARE *generate project-specific recurring fixes that current general automated program repair techniques (e.g., RSRepair, PAR and History-Driven) cannot generate?*

RQ2 Correctness, Quality, and Completeness: *Can* APARE *learn fix patterns automatically and use them to generate correct and high-quality fixes for recurring bugs? How complete are* APARE*-generated fixes compared to developer fixes?*

RQ3 Robustness: *Can* APARE *avoid generating incorrect fixes for non-recurring bugs?*

4.5.1 Collecting Recurring and Non-Recurring Bugs for Evaluation

To evaluate *APARE*, the evaluation includes 20 recurring bugs and 5 non-recurring bugs from Eclipse SWT, Eclipse JDT, ZK Web Framework, OpenJPA and Wicket. The bugs are *randomly* selected based on the following steps.

For *recurring* bugs, one (I or one of the co-authors, see "Statement of Contributions") examined the *randomly* sampled pairs of recurring fixes from the empirical study (§4.3), and read all the corresponding bug reports. All true recurring bugs that can be reproduced are kept for evaluation. First, if a pair of recurring fixes are from the same commit, the corresponding bug is excluded from our study. While *APARE* can still generate method fixes for those bugs, it is less likely for those fixes to help developers. Second, only the fixes that fix *true* bugs are used for evaluation, not adding new features or others. Many bug reports are not bugs [47]. I and my co-authors manually read these bug reports to identify true bugs. Third, many bug reports have no clear descriptions and cannot be reproduced. One tried his/her best to reproduce true bugs by reading the bug reports and the relevant code and fixes. Fourth, for a pair of recurring fixes, the bug whose fix was committed later, is reproduced so that *APARE* can use the pattern learnt from the earlier fix to fix the later bug automatically. In only two cases, the later bug cannot be reproduced because the bug report does not contain enough information, in which case, the earlier bug was reproduced.

For *non-recurring* bugs, we *randomly* sample bug reports from bug databases and use the second and third steps described above to find reproducible true bugs. In addition, the 60% matching approach in the empirical study (Section 4.3.2) is used to check if there are similar fixes in the past. If no similar past fix is identified, this non-recurring bug is kept for evaluation. If a potentially similar past fix is identified, the past fix is manually checked to see if they are indeed similar. If not, this bug is non-recurring and remains for evaluation.

The evaluated techniques (*APARE*, RSRepair) require passing test cases and at least one failing test case. I collect passing test cases written by developers and failing test cases from bug reports. If the bug reports do not provide failing test cases to expose the bug, we write failing test cases based on our understanding. Reproducing the bugs as described in the bug reports is a common and necessary practice for bug understanding and fixing [52, 21]. For ZK, the fault localization tool cannot capture the execution traces within the web-server environment, so fault localization information is manually collected

Table 4.3: Characteristics of the 20 Recurring Bugs

ID	Bug ID	Method Fixes by Developers	Recur. Method Fixes	Tests (Fail/Pass)	Methods from Fault Localization
1	SWT-139329	11	8	1/5290	148
2	SWT-94003	7	4	1/5165	183
3	SWT-102481	5	5	2/4960	47
4	SWT-91317	9	5	1/5293	63
5	SWT-71975	5	1	1/4839	126
6	JDT-97809	2	1	1/57	334
7	JDT-77510	1	1	1/130	196
8	JDT-109963	1	1	1/244	338
9	JDT-81244	1	1	1/49	148
10	JDT-99903	1	1	6/90	323
11	JDT-83499	1	1	1/44	54
12	JDT-111812	1	1	1/32	89
13	ZK-1227	1	1	N/A	1
14	ZK-1512	2	2	1/0	414
15	OpenJPA-1053	1	1	1/3	39
16	OpenJPA-819	1	1	1/32	317
17	Wicket-4290	20	2	1/30	43
18	Wicket-4487	10	1	1/36	25
19	Wicket-4829	17	5	1/32	60
20	Wicket-4012	4	1	1/34	168

either from the diagnosis information in bug reports or the execution traces of failing test case.

Table 4.3 shows the details of the 20 recurring bugs for evaluation. For each bug, it shows the total number of method fixes by developers in one commit, and among these, the number of recurring method fixes. The number of recurring method fix is the maximum number of correct method fixes that we expect *APARE* to generate. Next, it shows the number of failing/passing test cases used by fault localization, and the total number of methods that are identified by fault localization.

RQ1 Complementarity: *Can* APARE *generate project-specific recurring fixes that current general automated program repair techniques (e.g., RSRepair, PAR and HistoryDriven) cannot generate?*

Motivation. This evaluation compares *APARE* with general program repair tools because 1) there are no specialized repair techniques for project-specific recurring fixes; and 2) it is unclear whether general repair tools can also fix recurring bugs effectively. This evaluation shows that APARE can fix bugs that general program repair tools cannot. However, this

evaluation should not be interpreted as APARE can fix more bugs than existing tools for general bugs. Rather, the evaluation shows that APARE complements existing techniques by targeting project-specific recurring fixes.

Approach. Many automated program repair techniques explicitly target certain types of bugs. Particularly, constraint-solving based techniques such as Angelix and SemFix, repair defective conditions. Project-specific recurring fixes that *APARE* specializes are unlikely to be repaired by modifying conditions. In fact, most of the recurring bugs evaluated in this work cannot be repaired by altering the execution by modifying conditions. In contrast, it is unclear whether search-based repair techniques can effectively generate project-specific recurring fixes. Thus, this evaluation chooses to compare *APARE* with a search-based technique–RSRepair, which is shown to be more effective than the classic search-based technique–GenProg, to see if RSRepair is capable of generating fixes with rich project-specific semantics. In addition, a best case analysis is performed to compare *APARE* with a pattern-based technique–PAR and a history-driven technique–HistoryDriven.

Below I describe how the comparison is conducted.

RSRepair: Since the existing implementation of RSRepair only repairs C programs, RSRepair is reimplemented for repairing Java bugs. The implementation is functionally equivalent to RSRepair. For a fair comparison, the same test cases and fault localization information are used for both *APARE* and RSRepair. In addition, this evaluation uses the same setup that is described in the RSRepair paper: 40 random mutants are produced in each generation, and totally 10 generations to produce a total of 400 mutants for each bug. RSRepair is ran 10 times on each bug, which generates a total of 4,000 mutants for each bug. We consider a fix generated by RSRepair **correct** if it indeed fixes the bug.

PAR: A best case analys is performed since the source code of PAR is not available. In order for PAR to generate a complete fix, all the fix ingredients to form the fix must be covered by the fix patterns. Particularly for PAR, one fix ingredient refers to one fix pattern used by PAR; a fix from PAR may be composed by one or multiple fix ingredients (i.e., fix patterns). Therefore, I examine the fixes of all the 20 recurring bugs and check if one fix can be formed by one or multiple PAR's fix patterns. If so, it is considered that the bug can be fixed by PAR. This estimation is the best case for PAR because even if all fix ingredients of one fix are included by PAR, there is no guarantee that PAR generates the fix since it depends on PAR's search ability.

HistoryDriven: Similar to the comparison with PAR, a best case analysis is performed for a history-driven technique by Le et al. [68]. HistoryDriven is different from PAR in two aspects. First, HistoryDriven uses a set of mutation operators (i.e., insert, replace, and delete operators from GenProg, a few mutation testing operations, and a subset of fix

patterns from PAR) to form all patch candidates. Second, HistoryDriven ranks the patch candidates in the search space by employing a similarity score to measure the similarity among frequent fix patterns (i.e., mined from a large repository of past fixes). Despite such differences, in order to form the correct fix, similar to PAR, the fix ingredients must be covered by one or multiple of the mutation operators by HistoryDriven. Thus, we are able to conduct a best case analysis (similar to that for PAR) for HistoryDriven as well. I examine the fixes of all the 20 recurring bugs and check if one fix can be formed by one or multiple mutation operators of HistoryDriven. If so, the bug can be fixed by HistoryDriven. HistoryDriven contains three sets of mutation operators: GenProg mutation operators, mutation testing operators and PAR mutation operators. The later two are pattern-based operators, and are straightforward to check. The two mutation operators from GenProg, i.e., *insert* and *replace* statements, require correct statements from the buggy version to be inserted or replaced with. Search the correct statements in the same file is performed by ignoring white spaces. If such correct statements exist in the buggy file, by this best case analysis, the *insert* or *replace* operators of HistoryDriven are towards forming a correct fix.

Results. Table 4.4 shows the results of applying *APARE* on the 20 recurring bugs. For *APARE*, we show five columns. The first column shows the number of fix patterns learnt. For *APARE*, it is the number of fix patterns learned from past fixes after filtering (§4.4.1). Columns 'Precision', 'Recall' and 'Completeness' of Table 4.4 are defined as follows, where the definitions of precision and recall are standard: **precision** is the portion of method fixes generated that are correct; **recall** is the portion of *recurring* method fixes that are generated correctly; and **completeness** is the portion of developer fixes generated correctly (measured at the line level). Column 'RSRepair' shows whether RSRepair can generate a correct fix for each bug. For the fixes generated by RSRepair that make all test cases pass, I manually examine whether they are correct.

RSRepair: RSRepair cannot be applied on four of the bugs for the following reasons. First, SWT 71975 and 94003 are two GUI bugs. RSRepair is unable to automatically determine whether a test case fails or passes for these bugs because it cannot programmatically verify the graphical state affected by the bug. Conceivably, we could manually verify the GUI state for each mutant generated by RSRepair. However this is infeasible, since 4000 mutants are generated per bug, meaning up to 4000 GUI states would need to be manually inspected. Second, RSRepair can not run on the two ZK bugs because RSRepair needs execution traces from test cases. However, the fault localization tool cannot capture execution traces within the web server environment. In contrast, *APARE* generates 7 correct method fixes for these four bugs.

Therefore, RSRepair is applied on the remaining 16 recurring bugs. For these bugs, *APARE* generates 13 correct method fixes. For two of the 16 bugs, RSRepair generates

fixes that pass all test cases; however, upon manual inspection, the two fixes, which are equivalent to deleting functionalities, are incorrect and do not fix the bug. This finding complies with a recent study about plausible fixes [117].

PAR: PAR cannot generate fixes for the 20 methods that *APARE* generates correct fixes for. PAR may generate correct fixes for three methods that *APARE* cannot by using two fix patterns of PAR ('adding null checker' and 'parameter replacer'). The fix pattern 'adding null checker' adds a null checker to prevent null pointer exceptions. 'Parameter replacer' fix pattern replaces a parameter of a method call with a type compatible variable or expression.

HistoryDriven: HistoryDriven cannot generate fixes for the 18 methods that *APARE* generates correct fixes for (*APARE* generates 20 in total). HistoryDriven may generate fixes for 6 methods that *APARE* cannot by using a combination of several mutation operators (i.e., 'replacing method call parameter', 'delete statement', and 'change type cast').

In summary, current search-based and pattern-based techniques RSRepair and PAR are ineffective in repairing the 20 recurring bugs: RSRepair repairs none and PAR potentially could repair three faulty methods at the best case. Therefore, *APARE* complements general automated repair techniques by specializing project-specific recurring fixes.

RQ2 Correctness, Quality, and Completeness: *Can* APARE *learn fix patterns automatically and use them to generate correct and high-quality fixes for recurring bugs? How complete are* APARE*-generated fixes compared to developer fixes?*

Motivation. *APARE* is applied on the 20 recurring bugs to assess whether *APARE* can fix recurring bugs with acceptable precision, recall and completeness level. Reasonable precision and recall are important for automated program repair techniques. In addition, the completeness of the fixes generated by *APARE* is evaluated because although *APARE* generates complete fixes for the faulty methods, the generated recurring fixes maybe partial to the complete fix that fixes the bug as some of the complete fix may not be recurring.

Approach. Given a bug, *APARE* uses spectrum-based fault localization to generate a list of possible faulty methods. For a fair comparison, the same list of faulty methods that is used by RSRepair is used by *APARE* too. *APARE* then generates fix patterns automatically from all the fixes committed before the developer fixes for the given bug, and applies the novel techniques described in §4.4 to generate method fixes automatically. A fix that is generated by *APARE* is **correct** if it indeed fixes the bug under repair. In addition,

Table 4.4: Results on the 20 Recurring Bugs. Column 'Repair?' shows whether a bug is repaired by *APARE* or RSRepair. ✔✔ indicates that a bug is fixed correctly and completely by an approach. ✔ means *APARE* generates correct method fixes. ✘ denotes that an approach failed to fix a bug. Column 'Patterns' shows the number of fix patterns used by *APARE*. Column 'Comp.' shows the completeness. The completeness and recall of RSRepair is 0. 'AVG' in the last row is short for 'Average'.

ID	***APARE***					**RSRepair**	
	Patterns	Precision	Recall	Comp.	Repair?	Recall	Repair?
1	1,061	4/6 (66.7%)	4/8 (50%)	66/87 (75.9%)	✔	0/0	✘
2	374	8/9 (80%)	4/4 (100%)	24/154 (15%)	✔	—	—
3	530	2/4 (50%)	2/5 (40%)	32/78 (41%)	✔	0/2	✘
4	1,418	2/3 (66.7%)	2/5 (40%)	10/32 (31.3%)	✔	0/0	✘
5	46	0/0 (N/A)	0/1 (0%)	0/27 (0%)	✘	—	—
6	793	1/1 (100%)	1/1 (100%)	3/4 (75%)	✔	0/0	✘
7	327	0/0 (N/A)	0/1 (0%)	0/8 (0%)	✘	0/0	✘
8	929	1/1 (100%)	1/1 (100%)	6/6 (100%)	✔✔	0/0	✘
9	440	1/1 (100%)	1/1 (100%)	8/8 (100%)	✔✔	0/0	✘
10	902	0/0 (N/A)	0/1 (0%)	0/13 (0%)	✘	0/0	✘
11	426	0/1 (0%)	0/1 (0%)	0/5 (0%)	✘	0/0	✘
12	998	0/0 (N/A)	0/1 (0%)	0/5 (0%)	✘	0/0	✘
13	176	1/1 (100%)	1/1 (100%)	6/6 (100%)	✔✔	—	—
14	204	2/3 (66.7%)	2/2 (100%)	10/10 (100%)	✔✔	—	—
15	507	1/1 (100%)	1/1 (100%)	19/19 (100%)	✔✔	0/0	✘
16	273	0/0 (N/A)	0/1 (0%)	0/6 (0%)	✘	0/0	✘
17	98	0/0 (N/A)	0/2 (0%)	0/151 (0%)	✘	0/0	✘
18	647	1/1 (100%)	1/1 (100%)	3/120 (1.7%)	✔	0/5	✘
19	711	0/0 (N/A)	0/5 (0%)	0/0 (0%)	✘	0/0	✘
20	587	0/2 (0%)	0/1 (0%)	0/27 (0%)	✘	0/0	✘
AVG	572	24/34	20/44	185/772		0/7	
		70.6%	45.5%	24.0%		0%	

whether the fix is **equivalent** to the developer fix is also assessed. Developer fixes are used for evaluation only, and are not required for applying *APARE*.

Results. *APARE* is applied to repair the 20 recurring bugs under two settings: spectrum-based fault localization and perfect fault localization information. First, using spectrum-based fault localization results, *APARE* achieves 70.6% precision and 45.4% recall on average (Table 4.4). The recall is low, partially because the design of *APARE* favors high precision. Automated program repair is challenging; PAR, SPR and GenProg fix only 1–17% bugs, which is still very valuable nonetheless [61, 70, 117]. As shown in Columns 'Repair?' of Table 4.4 (see **RQ1**), most of the method fixes by *APARE* cannot be generated by the current techniques.

Second, *APARE* is evaluated given *perfect* fault localization information, which are the methods that developers fixed. Table 4.5 shows the precision, recall, and completeness when *APARE* is given correct and complete fault localization information, the methods that developers fixed. This setting evaluates the effectiveness of *APARE* when independent of fault localization techniques. If developers know which methods are faulty or fault localization research advances, *APARE* could have generated more accurate fixes for more methods. The overall precision increases to 87.1%, and the recall increases to 53.5%. In total, *APARE* generated 31 method fixes for the 20 recurring bugs. Among them, 27 method fixes are correct.

Table 4.4 also shows that *APARE* generated complete fixes for 5 bugs and generated over 75% complete fixes for an additional 2 compared to the complete fixes by developers. Although the generated method fixes do not always completely fix the bugs, they should help developers generate the complex fixes, because they are of high quality. Specifically, among the 24 correctly generated method fixes, 20 of them are identical to developer fixes. These fixes should be of high quality based on the definitions of previous studies [61, 135], which should help developers improve debugging correctness [135]. The manual examination of the 4 correct patches that are different from developer fixes show, that these 4 patches are easy to understand and should also be of high quality. A user study to rate the quality of these fixes remains as future work.

RQ3 Robustness: *Can* APARE *avoid generating incorrect fixes for non-recurring bugs?*

Motivation. When *APARE* is applied to fix a bug, *APARE* does not have the knowledge regarding whether the bug is recurring or not. There are two possible scenarios: 1) Developers may know the bugs to fix are recurring bugs, then developers will only apply

Table 4.5: Results on the 20 Recurring Bugs with Correct and Complete Faulty Locations

ID	Bug ID	Precision	Recall	Completeness
1	SWT-139329	4/5 (80%)	4/8 (50%)	69/87 (79.3%)
2	SWT-94003	8/8 (100%)	4/4 (100%)	24/160 (15%)
3	SWT-102481	4/6 (66.7%)	4/5 (80%)	62/78(79.5%)
4	SWT-91317	2/2 (100%)	2/5 (40%)	8/32 (25%)
5	SWT-71975	0/0 (N/A)	0/1 (0%)	0/27 (0%)
6	JDT-97809	1/1 (100%)	1/1 (100%)	3/4 (75%)
7	JDT-77510	0/0 (N/A)	0/1 (0%)	0/8 (0%)
8	JDT-109963	1/1 (100%)	1/1 (100%)	6/6 (100%)
9	JDT-81244	1/1 (100%)	1/1 (100%)	8/8 (100%)
10	JDT-99903	0/0 (N/A)	0/1 (0%)	0/5 (0%)
11	JDT-83499	0/0 (N/A)	0/1 (0%)	0/5 (0%)
12	JDT-111812	0/0 (N/A)	0/1 (0%)	0/5 (0%)
13	ZK-1227	1/1 (100%)	1/1 (100%)	6/6 (100%)
14	ZK-1512	2/3 (66.7%)	2/2 (100%)	10/10 (100%)
15	OpenJPA-1053	1/1 (100%)	1/1 (100%)	19/19 (100%)
16	OpenJPA-819	0/0 (N/A)	0/1 (0%)	0/6 (0%)
17	Wicket-4290	0/0 (N/A)	0/2 (0%)	0/151 (0%)
18	Wicket-4487	1/1 (100%)	1/1 (100%)	3/120 (1.7%)
19	Wicket-4829	1/1 (100%)	1/5 (20%)	3/112 (1.8%)
20	Wicket-4012	0/0 (N/A)	0/0 (0%)	0/27 (0%)
	Total/Average	**27/31 (87.1%)**	23/43 (53.5%)	248/772(32.1%)

Table 4.6: Results on the 5 Non-Recurring Bugs

Bug ID	Methods Identified by Fault Localization	Patterns (After Filtering)	Fixes by *APARE* (0 is the best)
SWT-90258	393	474	0
SWT-118659	116	1176	0
JDT-82253	4	519	0
JDT-85262	323	773	2
ZK-1008	1	139	0

APARE for recurring bugs; and 2) Developers do not know whether the bugs to fix are recurring bugs. For the 2) scenario, *APARE* should be able to avoid generating incorrect patches for *non-recurring* bugs. Since *APARE* is designed for recurring fixes only, non-recurring bugs are error inputs for *APARE*.

Approach. *APARE* is applied to the 5 non-recurring bugs. Section 4.5.1 describes how the 5 non-recurring bugs are collected. Then, the patches generated by *APARE* are manually checked to determine whether they are correct.

Results. Table 4.6 shows the details of the 5 evaluated non-recurring bugs. *APARE* generates only two incorrect patches for one Eclipse JDT bug and avoids generating incorrect ones for the other four. I manually check the two false positives and find them easy to understand so it should be easy for developers to quickly filter them out.

4.6 Discussions and Threats to Validity

4.6.1 Execution Time

The execution time of *APARE* consists of two parts: 1) fault localization time: the time to run test cases to obtain fault localization results; and 2) patch generation time: the time to extract fix patterns and perform matching between fix patterns and the identified target methods. The fault localization time is required by both *APARE* and existing techniques such as RSRepair and PAR, which takes from seconds to hours depending on the time to run all test cases of a project. Compared to search-based approaches such as RSRepair, *APARE* is faster because its time in patch generation is shorter. Particularly, *APARE* saves time because *APARE* does not conduct patch validation (i.e., whether the patched version passes all the test cases), which is used by RSRepair to significantly reduce the number of incorrect patches while searching in a sparse searching space. The experiment is conducted on an 3.1GHz Intel Core i5 machine with 8 GB memory. *APARE*'s patch generation time

is on average 65 minutes per bug for Eclipse JDT, and 75 seconds per bug for all other evaluated projects. Eclipse JDT bugs take much more time to repair because often Eclipse JDT code has complex control flows which makes it expensive to compute control and data dependencies. In the future, all past fixes can be preprocessed and the learnt fix patterns can be stored in a database to save time for patch generation. In contrast, RSRepair's patch generation time ranges from 4–30 hours per bug, much longer than that of *APARE*, due to the large search space.

4.6.2 Threats to Validity

APARE specializes in generating correct and high quality fixes to fix recurring bugs. *APARE* cannot fix non-recurring bugs, but it can avoid generating incorrect fixes for non-recurring bugs. Since automatically fixing all bugs is almost impossible, I believe in focusing on certain types of bugs, i.e., recurring bugs in this paper. It is quite feasible for developers to combine *APARE* and existing program repair techniques to generate more fixes.

APARE extracts bug fixing commits from software repository based on keywords and links to bug reports in commit messages. Therefore, *APARE* relies on the quality of the commit messages to obtain an accurate and complete set of past fixes, which means *APARE* may not work well for projects with low-quality commit messages. Also, *APARE* may not work for projects with short development histories since *APARE* focuses on generating project-specific recurring fixes; for projects with short development histories, other automated program repair techniques that use general patterns across projects (e.g., PAR) may be applicable.

In order to compare *APARE* with RSRepair on the same set of bugs from Java projects, RSRepair for Java programs (RSRepair-J) is implemented while the original RSRepair [116] is for C programs (RSRepair-C). RSRepair for Java is re-implemented by carefully reading the source code of RSRepair in detail and implement that for Java accordingly. However, there is no way to guarantee the absolute equivalence of the RSRepair-J and RSRepair-C due to multiple reasons, such as differences of the underlying AST parsers for different languages (JDT for Java and CIL for C).

4.7 Chapter Summary

This chapter presents *APARE*, which learns fix patterns from past bug fixes automatically, and uses the fix patterns to generate bug fixes for new recurring bugs automatically. The

evaluation on five programs shows that *APARE* can generate accurate and high quality bug fixes. *APARE* complements existing automated program repair techniques because it can fix many bugs that existing automated program repair techniques cannot fix. In addition, this chapter presents an empirical study of recurring fixes, the findings of which guided the design of *APARE* to identify recurring fixes with similar contexts.

Chapter 5

Priv: Prioritizing, Visualizing and Fixing Vulnerability Warnings of Static Application Security Testing

5.1 Introduction

Static analysis techniques are widely used in practice to ensure the quality of the software[**?**, 17]. In particular, developers often rely on static application security testing (SAST) techniques to detect security vulnerabilities. Unlike dynamic analysis, SAST is able to detect potential vulnerabilities that remain uncovered after in-house testing. Hence, SAST has been shown to be effective in improving the security of applications [79, 54, 121].

Aside from research prototypes[79, 42, 86], there are many popular commercial SAST products, such as AppScan Source [48], CheckMark [2], and Fortify [3]. However, as shown in previous studies [53, 26], developers often encounter many challenges when using static analysis techniques, which results in developers' underuse of SAST techniques. We have been working with security experts from AppScan Source for the past four years and, together with the security experts, we have identified several limitations of SAST techniques.

First, SAST techniques usually detect a large number of vulnerability warnings with a high false positive rate. Current SAST techniques help reduce the impact of false positives by grouping the detected warnings based on some categories (e.g., severity, vulnerability type, etc) [53]. Unfortunately, such categorization does not provide sufficient support to developers in tackling the large number of vulnerability warnings and prioritizing developers'

quality assurance efforts [53, 26]. Second, current SAST techniques lack of support in identifying actionable vulnerability warnings (i.e., warnings that are both true vulnerabilities and require a fix)[137]. Finally, SAST techniques do not provide sufficient guidance on how to fix the vulnerability warnings. SAST techniques may provide a general guidance (referred as remediation pages) for each type of vulnerability warnings (e.g., one remediation page for all SQL injection attacks). Such general remediation pages only provide high-level information, but the needed specific information may vary for each detected warning.

Therefore, to better assist developers in fixing the detected vulnerabilities, this chapter proposes a set of approaches that specifically help developers with fixing vulnerability warnings that are detected by SAST. The proposed approaches are implemented as a tool, named *Priv*. *Priv* is well-received by security experts and is under integration into industrial practice, and helps AppScan Source developers and users with fixing security vulnerability warnings. *Priv* includes two phases. Phase I provides a global-view visualization that helps prioritize developers' quality assurance effort. *Priv* first identifies suggested fix location for each detected vulnerability warning, and uses data visualization to illustrate the vulnerability warnings in clusters, based on shared suggested fix locations. Phase II supplements additional essential information for efficient remediation, which includes identifying actionable vulnerability warnings, and generating customized remediation pages for each detected warning, which includes the buggy code and the corresponding fix code.

Priv targets the most common and the most prominent vulnerability types: cross-site scriptings (XSS), SQL injections (SQLi), path traversals (PATHtrv), command injections (COMMi), and second-order injections. A prior study [129] shows that cross-site scriptings, SQL injections, and parameter tampering account for more than a third of Web application attacks. Cross-site scriptings attack systems by executing malicious scripts on victim's browser. SQL injections is caused by unsanitized inputs that purposely damage databases. Second-order injections are closely related to SQL injections. Second order injections arise when unsanitized user inputs are stored in the system, and using the inputs to access the database causes damages to the system. Command injection is an attack that executes commands from untrusted input on the server side.

Priv is developed and integrated into a mature commercial SAST product–AppScan Source. *Priv* is evaluated on one closed source (i.e., AltoroJ, an internal testing application for evaluating AppScan Source) and five open-source web applications that are commonly used for study vulnerabilities: WebGoat, JavaVulnerable Lab, Vulnerable Web, Bodgeit, and HeisenBerg. The evaluation of *Priv* answers the following research questions:

RQ1: How do the suggested fix locations by *Priv* compare to the ones that are identified by developers?

The evaluation shows that for 50–100% of the detected vulnerability warnings, *Priv*'s suggested fix locations are identical to the ones identified by developers.

RQ2: How many actionable warnings can *Priv* find for better diagnosis?
The evaluation shows that *Priv* identifies 4–2170 actionable database- or attribute-related vulnerability warnings. Without *Priv*, the initial false positive rate would be up to 88.6%; With *Priv*, the false positive rate is reduced to 0%.

RQ3: What is the quality of the *Priv*'s automatically-generated fixes?
The evaluation shows that *Priv* generates complete fixes for many detected vulnerability warnings.

5.2 Background on SAST and AppScan Source

This section provides background knowledge on static application security testing (SAST), which applies static analysis techniques to detect vulnerabilities in source code. This section also describes details of AppScan Source—one of the most prominent commercial SAST products in which *Priv* is integrated.

Using Static Analysis to Detect Vulnerabilities. Static analysis, in particular taint analysis is used to statically detect vulnerabilities in information flow (i.e., how a tainted object is passed throughout the program) [79]. The tainted object propagation consists of a source, a sink, and the data-flow ("reachability") from the source to the sink. An information-flow vulnerability typically starts with obtaining an untrusted input (i.e., source) and finally reaches to statements that perform security-critical functionalities (i.e., sink), such as executing SQL queries. For example, SQL injections are caused by unsanitized input being passed to databases for potentially malicious activities. The untrusted input could come from web application APIs (e.g., through a method call `javax.servlet.http.HttpSession.getAttribute` in Java), and passed to database execution APIs (e.g., `java.sql.Statement.executeUpdate`). In addition, tainted analysis is applied in mobile security to protect users' confidential information from being leaked [16].

AppScan Source. AppScan Source [48] is one of the leading commercial products in applying static analysis for security testing (i.e., vulnerability detection). With the first release back in 2003, the latest version of AppScan Source is Version 9.3, which is the version that *Priv* is integrated to. Note that although *Priv* is integrated to AppScan Source, we believe *Priv* is general and can be applied to other SAST products due to the similarities among SAST products.

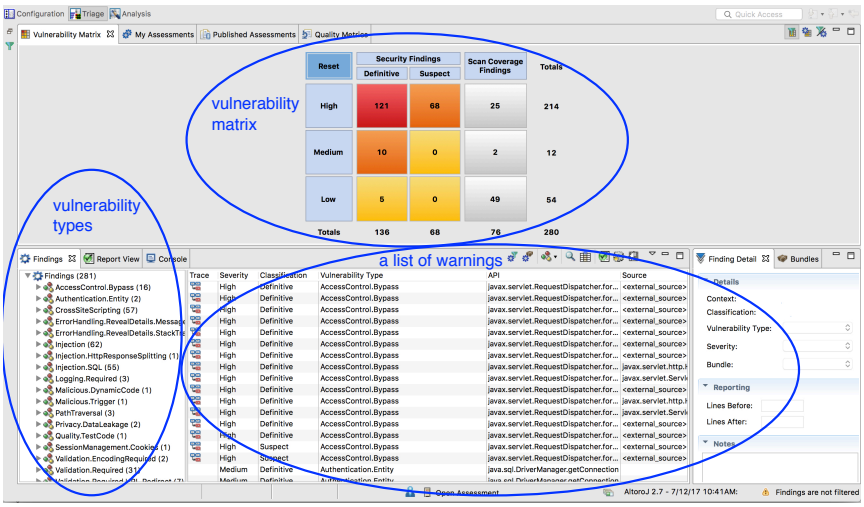

Figure 5.1: A screenshot of the Report View of AppScan Source.

Figure 5.1 shows a screenshot of the result of applying AppScan Source to the evaluated projects (AltoroJ). In the screenshot, the top component shows the vulnerability matrix, which provides multi-dimensional prioritization based on severity level (i.e., high, medium, and low), and the confidence on the detected vulnerabilities (i.e., definitive, and suspect). 'Red' color highlights the number of vulnerabilities with the highest priority to fix. The bottom-left component shows a list of vulnerability types, such as 'Cross-Site Scripting'. The bottom-right component lists the detected vulnerability along with the key information, such as file name, line number, API, source, and sink.

When users click on one detected vulnerability, AppScan Source shows the detailed information as shown in Figure 5.2. Such detailed information includes the trace information (i.e., data-flow of a detected vulnerability), a window to show the code snippet, and a general solution on how to fix this type of vulnerability in a remediation page. All the information displayed for one vulnerability warning is stored in an assessment file in XML format. We built *Priv* by leveraging the information in assessment files.

An example of SQL injection is used to explain the representation of the trace in AppScan Source. In Figure 5.2, the tree on the top-left is the visualization of a trace. The trace contains two red nodes: the left node is the source, and the right node is the sink.

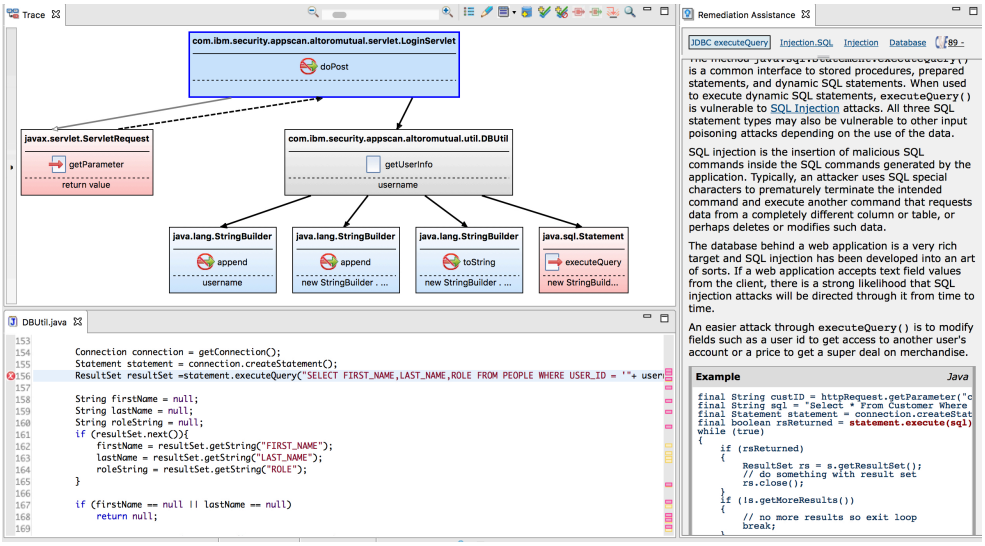

Figure 5.2: For each vulnerability warning, AppScan Source shows the trace (i.e., to visualize the data-flow from source to sink), a code window to show the code snippet of the highlighted node (i.e., clicked by the user) in the trace, and the current remediation page (e.g., text description, examples of buggy code and the corresponding fix).

The source uses the API `getParameter` from the class `javax.servlet.ServletRequest`, which takes user input from an HTTP request. Hence, the obtained input parameter value is considered as a malicious input. The sink executes SQL queries to communicate with the database (i.e., using `java.sql.Statement.executeQuery`). The other nodes in the trace show how the malicious input is propagated from the source to the sink. For example, the blue node at the root position (i.e., the function `doPost`) calls `getParameter` and passes the obtained parameter value to the function `getUserInfo`. Since the data flow between the source and the sink does not contain a validation method and the sink does not call `PreparedStatement`, this data flow is determined as a SQL injection.

AppScan Source may report warnings that are false positives due to the limitation of taint analysis, e.g., infeasible paths. High false positive rate is a common limitation shared by many static analysis techniques for bug detection, e.g., FindBugs.

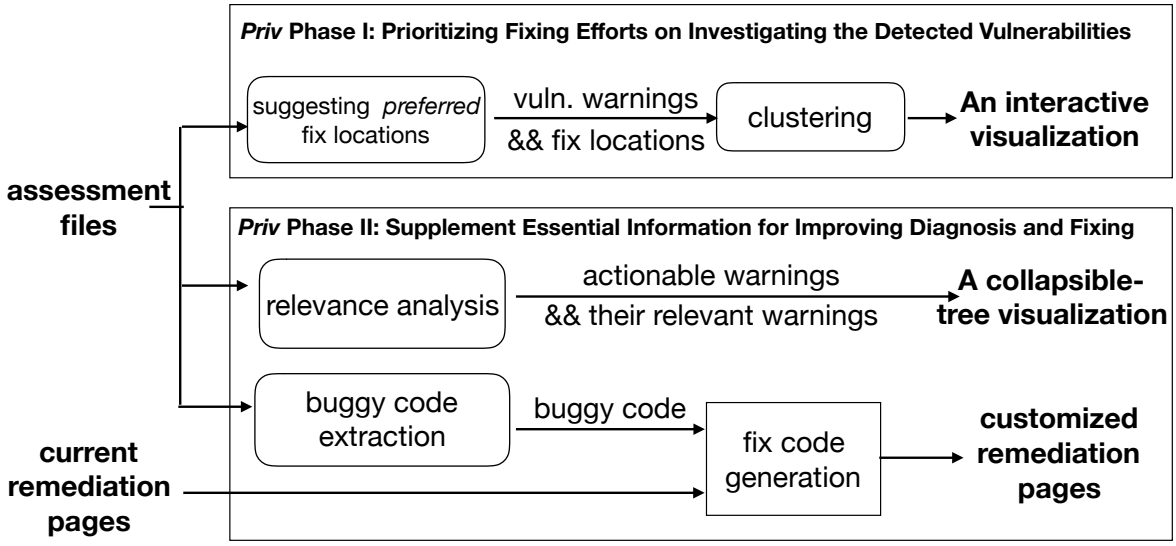

Figure 5.3: The Overview of *Priv*. The output 'force-directed graph' and 'collapsible tree' visualizations can be displayed in a browser.

5.3 The Design and Implementation of *Priv*

This section describes the design of *Priv* in detail. Figure 5.3 shows an overview of the proposed approach. *Priv* consists of two phases. Phase I (Section 5.3.1) contains two parts: 1) suggesting preferred fix locations (*pFixLoc*) for each detected vulnerability warning; and 2) grouping detected vulnerabilities into clusters based on shared *pFixLoc*. Phase II (Section 5.3.2) contains the components of identifying actionable warnings ('relevance analysis' in Figure 5.3), and producing customized remediation pages that includes 'buggy code extraction' and 'fix code generation' (as shown in Figure 5.3). The output of *Priv* includes a global-view interactive visualization using the force-directed graph to show clusters of warnings in which they share the same *pFixLoc*, a tree visualization to illustrate actionable warnings and their relevant warnings, and a customized remediation page showing automatically-generated suggested fix for each detected vulnerability warning.

5.3.1 Phase I: Prioritizing Fixing Efforts on Investigating the Detected Vulnerabilities

Suggesting Preferred Fix Locations

Given a detected vulnerability warning, there may be multiple places in the code that developers can add the corresponding fix (e.g., code validation code) to mitigate the security risk. Therefore, to minimize developers' efforts on fixing vulnerabilities, one needs to identify the most cost-effective fix location in the code, where adding a security fix can resolve the largest number of related vulnerability warnings. Such cost-effective fix locations are referred to as *preferred* fix locations (*pFixLoc*). *Priv* focuses on suggesting *pFixLocs* for four types of vulnerability warnings: SQL injection (SQLi), cross-site scripting (XSS), command injection (COMMi), and path traversal (PATHtrv). Below is the discussion on how *Priv* determines the *pFixLoc* for the studied vulnerability warnings.

The general fix strategy for the above-mentioned four types of information-flow vulnerabilities (i.e., untrusted input being executed) is to add code for validation in the information flow. Different validation code is used for different types of vulnerability warnings. For example, the validation to prevent PATHtrv restricts the access to certain directories or files. The validation to prevent XSS guards execution against malicious scripts.

For the purpose of *Priv*, which is to prioritize developers' fixing effort, there are two factors that *Priv* should consider when suggesting the *preferred* fix location for vulnerability warnings. The first factor is that the suggested *pFixLoc* should maximize fixing ability,

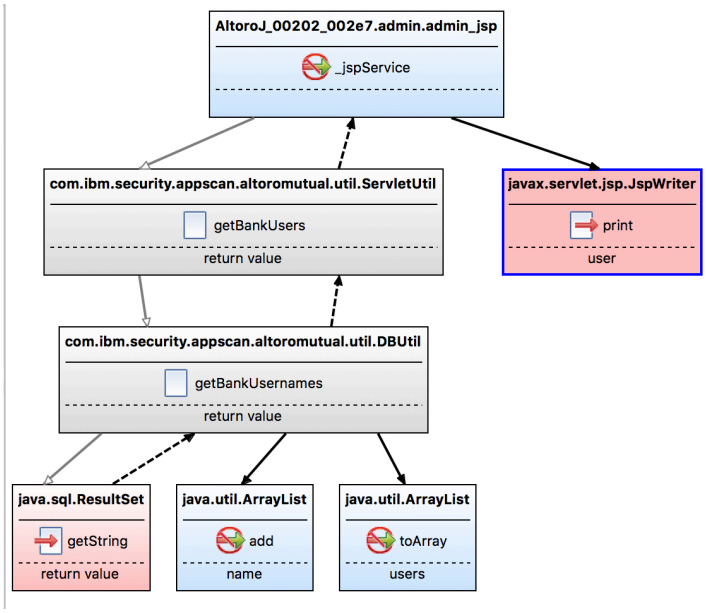

Figure 5.4: A Cross-Site Scripting Example to Illustrate the Possible Fix Locations.

which reflects the number of vulnerability warnings that can be fixed if adding the validation code at one *pFixLoc*, also referred to as "fixing multiple vulnerability warnings by fixing one location". The second factor is whether the suggested *pFixLoc* should minimize the potential interferences, which means that adding the validation code at this *pFixLoc* has minimal side effects, such as affecting other data-flow paths that also go through this *pFixLoc*. When the first factor is considered, *Priv* suggests *pFixLoc* that is close to the source to maximize the fixing ability. In addition to adding *pFixLoc* close to the source, security experts from AppScan Source point out that adding validations in Java classes (e.g., when initializing class fields) also helps improve fixing ability. The reason is that when untrusted input is used for initializing class fields of Java objects, developers may want to add customized validations based on specific class fields. When the second factor is considered, *pFixLoc* is preferred to be at the sink because the sink is unlikely to be shared with vulnerability warnings of different types. Thus, a validation that is specific to one type of vulnerabilities does not affect the vulnerability warnings of other types.

Priv uses a set of rules for suggesting *pFixLoc* for the four types of vulnerability warnings. The set of rules are determined based on the discussions with security experts from AppScan Source by considering the two above-mentioned factors: maximizing fix ability and minimizing potential interferences.

The strategy of suggesting *pFixloc* for the studied vulnerabilities considers all the following information without a particular order:

XSS

- Source;
- Direct caller of source;
- Java objects along the path from source to root caller;

SQLi, COMMi, and PATHtrv

- Sink;
- Direct caller of sink;
- Java objects along the path from root caller to sink;

The cross-site scripting example in Figure 5.4 is used to explain the reasons behind *Priv*'s choice of rules in suggesting *preferred* fix locations. Figure 5.4 shows the trace of an cross-site scripting warning that includes nodes and the data-flow among the nodes. The root caller (`_jspService`) gets untrusted input from the source (`getString()`) through `getBankUsers` and `getBankUsernames`. Then, the root caller outputs the unsanitized data, which may contain a malicious script that is sent to a victim's browser (`JspWriter.print()`). For this XSS, *Priv* suggests some locations (particularly the source, the direct caller of source, and Java objects in the path) in the path from the source to the root caller to maximize fix ability. *Priv* does not narrow down to only one *pFixLoc* and may suggest several *pFixLocs*, because it is possible that adding the validation code to the source does not fix multiple vulnerability warnings. *Priv* highlights a list of potential *pFixLoc* and delegates the final decision to developers. Note that identifying *pFixLoc* is the first and an intermediate step to group the SAST detected warnings for prioritizing the quality assurance effort.

For SQLi, COMMi, and PATHtrv, *Priv* suggests locations that are close to sink (i.e., on the path from root caller to sink) as *pFixLoc* (particularly the sink, the direct caller of the sink, and the Java objects along the path from root caller to sink) to minimize potential interferences. If the fix location is not close to the sink, it is possible that the validation (e.g., white-list style) for SQLi, COMMi, and PATHtrv, may interfere with vulnerability warnings of different types.

Grouping Vulnerability Warnings Based on Preferred Fix Locations

Priv groups vulnerability warnings based on *pFixLoc* into clusters to help developers prioritize fixing efforts. When developers investigate the warnings in one cluster instead of one-by-one, they become aware that working on one *pFixLoc* can fix the warnings in the same cluster. In addition, developers can choose to work on a particular cluster (e.g., the cluster with the most findings) for prioritize fixing efforts. Moreover, viewing the warnings in clusters enables developers to be aware of potential side-effects of placing validation methods by looking into the interferences among clusters. For example, if a white-list validation method is added to one fix location, and this fix location may also appear in the data flow of other warnings, the fix location is a potential interference point that developers should be aware of.

An Interactive Visualization for Prioritizing Fixing Efforts

Priv presents the clusters of warnings in an interactive visualization. The visualization provides a global view of all the SQLi, XSS, COMMi, and PATHtrv warnings that are detected by SAST in one project. Furthermore, the visualization allows developers to navigate any cluster of vulnerability warnings, or zoom in for one particular vulnerability warning.

The global-view visualization prioritizes the warning clusters that have the most significant size and/or more complex structures (e.g., with interferences with other clusters). Also, potential interferences among clusters are implicitly highlighted since clusters are designed to be distant from each other, so that the interferences among clusters are noticeable. Figure 5.6 shows an example of a single cluster. The blue node in the center is the shared fix location suggested by *Priv*. Each purple node represents one vulnerability warning. The orange nodes represent the rest of trace nodes that are not suggested as fix locations. The visualization is interactive so when developers click on one node, the details of the node are shown (Figure 5.5). The details include which file/class the suggested fix location is at, what methods the unsanitized input are propagated to, etc.

Priv applies an off-the-shelf graph drawing algorithm–force-directed graph drawing [131] to layout the global view of detected vulnerability warnings. Force-directed graph drawing algorithm is designed to simulate a physical system, in which nodes are either pulled close or pushed away based on forces assigned to them. Force-directed graph drawing algorithm requires no prior knowledge of the graph layout and tries to eliminate lines across different clusters of nodes. In the context of displaying clusters of vulnerabilities, the force-directed graph algorithm centers clusters around nodes that represent *pFixLocs* and assigns forces to

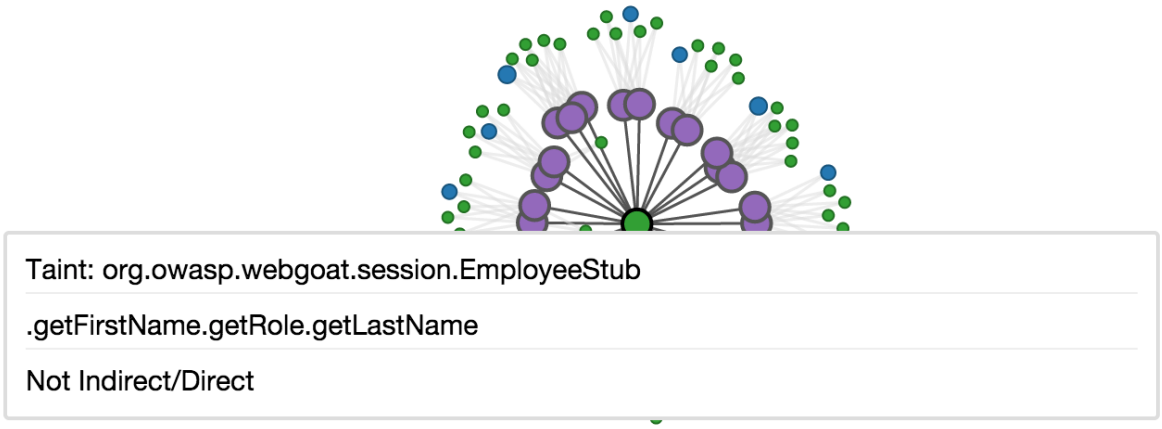

Figure 5.5: An Interactive Single Group Example grouped Force Directed Graph

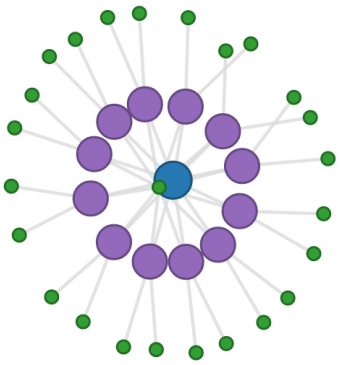

Figure 5.6: An Example of Warnings in a Single Cluster in Force Directed Graph

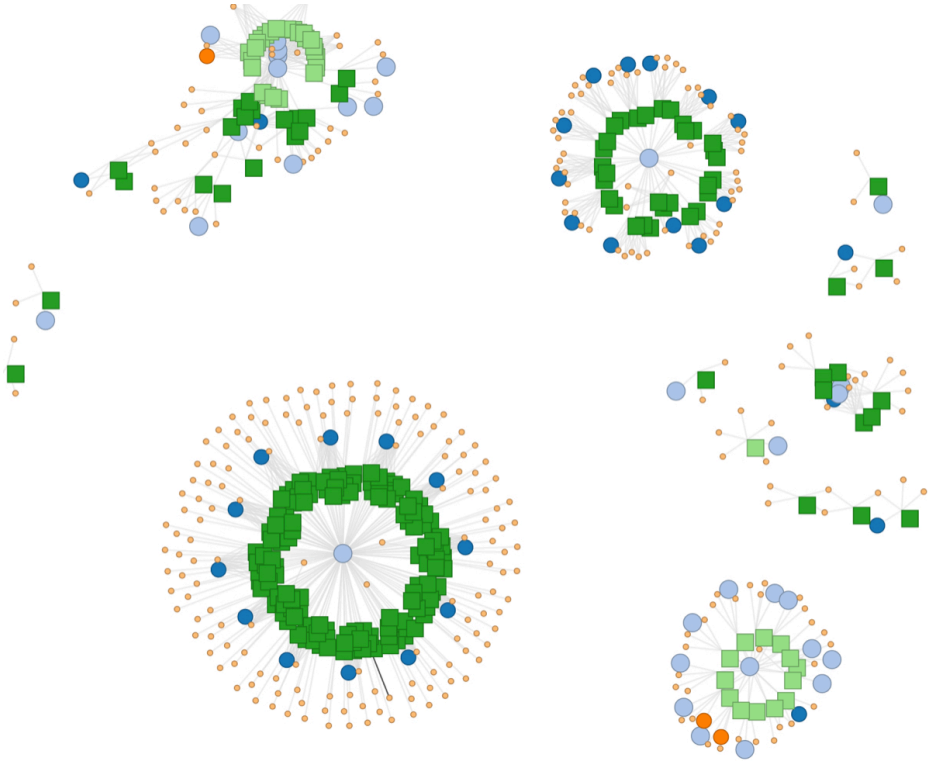

Figure 5.7: *Priv* provides the global-view visualization for WebGoat 5.3. Rectangles with various colors represent different types of vulnerability warnings. Circles present nodes in the trace: light blue circles are the preferred fix locations; dark blue circles are the *pFixLocs* that are from Java objects; orange circles represent the trace nodes other than *pFixLocs*.

the associated vulnerability warnings. Once the algorithm is finished, vulnerability warnings that have the same *pFixLoc* are clustered around the centered node of *pFixLoc* and are pushed away from the rest of the vulnerability warnings that do not have the same *pFixLoc*. Figure 5.7 shows the global-view visualization of WebGoat 5.3 that is generated by *Priv*. The visualization contains complex clusters (e.g., the top-left cluster) and large clusters (e.g., the bottom-left cluster). This global-view visualization guides developers to work on the large clusters for prioritizing fixing effort, and notifies developers that there are interferences among the complex clusters.

5.3.2 Phase II: Supplement Essential Information for Improving Diagnosis and Fixing

Identify Actionable Database/Attribute-Related Warnings

As shown in a previous study [53], high false positive rates add an extra burden to developers when fixing the warnings of static analysis techniques. Therefore, it is crucial to assist developers quickly to identify actionable warnings. However, unlike the typical false positives from static analysis techniques (e.g., caused by infeasible paths), another type of false positive may exist in SAST because SAST does not construct a complete information flow for stored injections (e.g., database/attribute-related warnings). These stored injections are real vulnerabilities only when there exists a complete information flow.

For example, second-order injection warnings are stored injections for which the malicious input remain dormant in the database. However, not every detected second-order injection is a real vulnerability. In the case when there is no code that reads the previously-stored malicious input from the database and passes it to an execution (named **exit points**), such second-order injections are neither **actionable** nor worth fixing. Namely, a second-order injection warning is actionable if the previously-stored data later is read from the database and then used by database queries (i.e., causing SQL injections) or in websites (i.e., causing cross-site scriptings).

Thus, it is crucial to provide developers extra information when identifying actionable warnings. *Priv* connects entry points and exit points on the same database tables to illustrate the complete information flows of second-order injections. First, *Priv* identifies **entry points** whose sinks write to databases and **exit points** whose sources read from databases. Second, *Priv* asks developers to specify the names of database tables that are accessed by each vulnerability warning of entry points and exit points. Finally, *Priv* connects entry and exit points that access the same database table. Note that it remains as future work for *Priv* to automatically extract the names of database tables, which is feasible. *Priv* is not able to automatically do so now because the current implementation of *Priv* has the restriction of using only the AppScan Source assessment files for independence and easy integration with AppScan Source.

Similar to second-order injections, *Priv* identifies *actionable* attribute-related warnings by connecting entry and exit points that are related to JSP attributes. *Priv* uses two sets of APIs to identify entry and exit points. Table 5.1 lists the two sets of APIs that are used by *Priv* to identify actionable attribute-related warnings. For example, *Priv* locates both the entry-point warnings whose sinks call `javax.servlet.http.HttpSession.setAttribute` to set the value of the attribute `attr`, and the exit-point warnings whose sources call

Table 5.1: APIs that are used to identify actionable attribute-related warnings.

API Set I	
entry point	javax.servlet.http.HttpSession.setAttribute
exit point	javax.servlet.http.HttpSession.getAttribute
API Set II	
entry point	javax.servlet.ServletRequest.setAttribute
exit point	javax.servlet.ServletRequest.getAttribute

`getAttribute` from the same Java package to get the value of the attribute `attr`. Then, *Priv* connects the two sets of vulnerability warnings (i.e., entry point and exit point) that are related to the attribute `attr` to provide developers with a complete information flow of the attribute `attr`. When connecting attribute-relevant warnings, *Priv* automatically extracts the values of attributes from source code using regular expression since the values of attributes are often string literals when passed to the APIs as parameters (e.g., `setAttribute` or `getAttribute`).

A Collapsible-Tree Visualization of Actionable Database- or Attribute-Related Warnings

Priv identifies actionable database- or attribute-related vulnerability warnings by connecting two sets of warnings that construct the complete information flow.The two sets of warnings are 1) entry points that write to databases or set attribute values of HTTP sessions/requests; 2) exit points that read from databases or get attribute from HTTP sessions/requests. Then, *Priv* presents the two sets of warnings in a collapsible-tree visualization to highlight actionable warnings.

Figure 5.8 shows an example of the collapsible-tree visualization that is provided by *Priv* to highlight actionable second-order injections of WebGoat [138]. The collapsible-tree visualization contains multiple layers. Figure 5.8 shows the look of the visualization when it is fully collapsed. Figure 5.8 highlights entry and exit points for the database table “employee". Note that a complete visualization may include several figures that are similar to Figure 5.8. In the complete visualization, the database tables are grouped and prioritized based on: 1) database tables with both entry and exit points; 2) ones with entry points only; and 3) ones with exit points only. In each of the above-mentioned groups, the tables are further sorted by the total number of associated entry and exit points.

The “employee” node can be expanded into two branches–entry and exit points. The

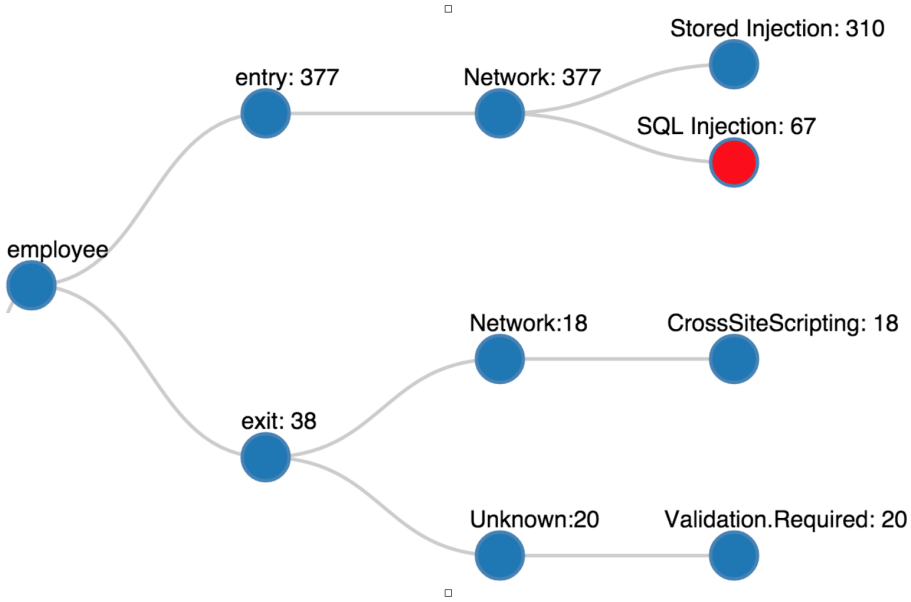

Figure 5.8: This visualization shows entry and exit points of database table “employee" in WebGoat 5.3.

layer next to the entry/exit points is the type of incoming sources or outgoing sinks, such as network or file system. This layer is useful since it provides a high-level understanding of what type of source makes the database table “employee” vulnerable. In addition, it helps developers understand that the malicious data enters from ‘network’ (e.g., reading from HTTP requests) and exits through “network” (e.g., a stored attack through cross-site scriptings).

The next layer is the vulnerability type. For example, the vulnerability type “SQL Injection” means that the entry points write to databases without proper validation methods, e.g., using “PreparedStatement”. “Stored Injection” refers to second-order injection, which means that “PreparedStatement” is used to prevent SQL injections; however, the warning may still be harmful since it may suffer from second-order injection attacks. The final layer (not shown in Figure 5.8) is to list the details of each warning.

Customizing Remediation Pages

SAST techniques may provide remediation pages to help developers understand each type of vulnerability and provide examples to show how to fix the vulnerabilities. Each vulnerability

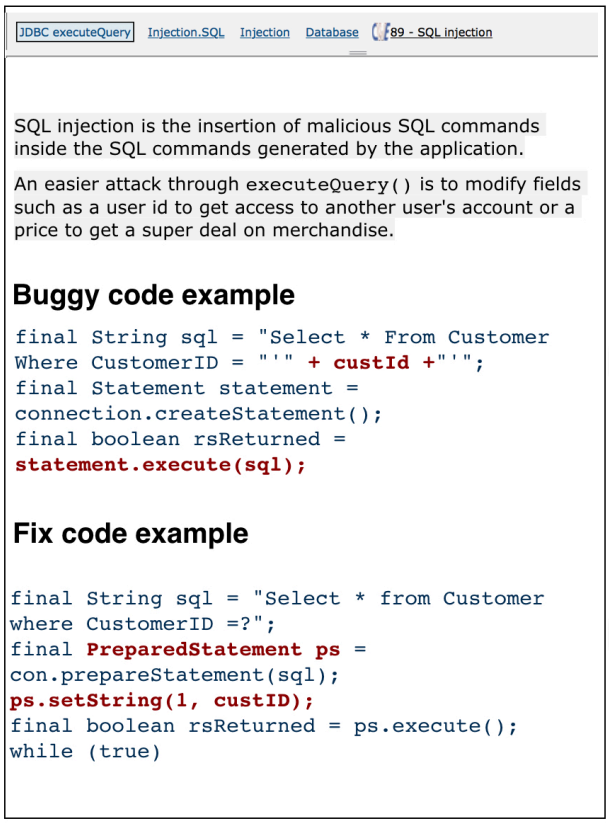

Figure 5.9: AppScan Source provides a general remediation page for all SQL injections. We simplified the test description, buggy code, and fix code for easier understanding.

type has one remediation page that is generalized to all the vulnerabilities of this type. Thus, the general remediation page does not provide customized diagnosis information (e.g., buggy code and the corresponding fix code) for each vulnerability since the remediation page only includes general examples. Therefore, *Priv* customizes remediation pages for each detected vulnerability warning for better diagnosis. For each detected vulnerability warning, *Priv* intercepts the current remediation page, and replace the general information with customized information on buggy code and fix suggestions. The current implementation of *Priv* offers customized remediation pages for SQL injections, cross-site scriptings, path traversal, command injections, and cryptography insecurity.

Figure 5.9 shows the existing remediation page of AppScan Source for SQL injections. *Priv* customized the current remediation page by replacing the general buggy and fix code, as shown in Figure 5.10. *Priv* re-constructs buggy code and generates the corresponding fix

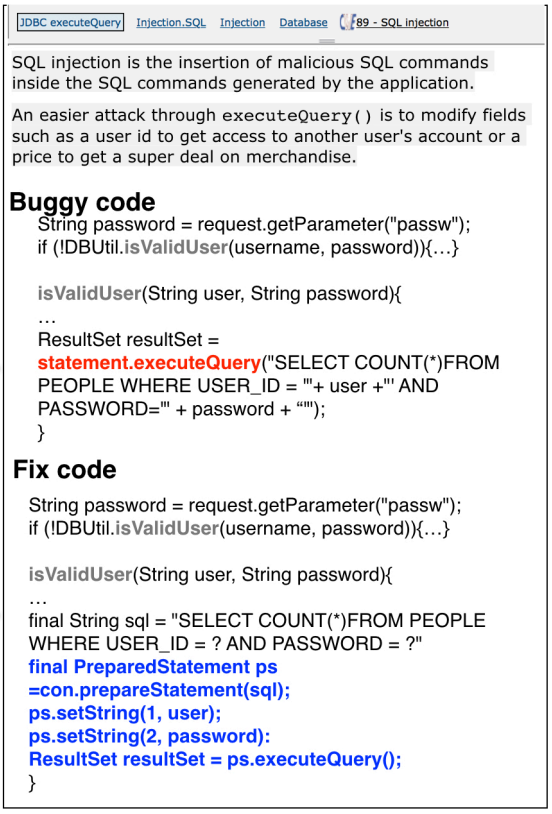

Figure 5.10: *Priv* replaces the general code examples in Figure 5.9 with the customized code snippets (as shown in the areas of buggy code and fix code).

```
<html>
- <&= request.getParameter("name")&>
+ <&=HTMLEntityEncoder.encode(request.getParameter("name"))&>
</html>
```

(a) XSS Fix Template

```
- ResultSet var1 = var2.executeQuery(``string_literal'' + var3);
+ final String sql = get_paramerized_query(``string_literal'');
+ final PreparedStatement ps = con.prepareStatement(sql);
+ ps.setString(1, var3);
+ ResultSet var1 = ps.executeQuery();
```

(b) SQL Injection Fix Template I

```
- String sql = ``string_literal'' + var1;
- ResultSet var2 = var3.executeQuery(sql);

+ String sql = get_parameterized_query(``string_literal'');
+ final PreparedStatement ps = con.prepareStatement(sql);
+ ps.setString(1, var1);
+ ResultSet var1 = ps.executeQuery();
```

(c) SQL Injection Fix Template II

```
+ if (validate(var2))
  var1 = Runtime.getRuntime().exec(var2);
```

(d) Command Injection Fix Template

```
+ if (validateWhiteList(var2))
  var1 = new File(var2);
```

(e) Path Traversal Fix Template

```
- var = MessageDigest.getInstance("MD5"|"SHA");
+ var = MessageDigest.getInstance("SHA-256");
```

(f) Cryptography Insecurity Fix Template

Figure 5.11: We concluded one fix template for fixing the vulnerabilities based on the current remediation page of AppScan Source.

code by analyzing the trace information in the assessment file (i.e., the files where AppScan Source stores the analysis results). Currently, fix templates are manually concluded from the current remediation pages, and so *Priv* can apply the concluded fix templates to generate fix code for each vulnerability. For example, the fix template is manually crafted (shown in Figure 5.11b) for fixing SQL injections based on the fix example shown in Figure 5.9. In the future, I plan to automate the process of manually crafting fix patterns from documentation. The buggy code in line 1 executes an unsanitized SQL query. The fix prevents the SQL injection attack by using `PreparedStatement`. Line 2 initializes the variable `sql` with a modified version of the SQL query used in the buggy code. For example, `string_literal` in line 1 is "SELECT COUNT(*)FROM PEOPLE WHERE USER_ID =", and then the variable `sql` in line 2 is assigned with "SELECT COUNT(*)FROM PEOPLE WHERE USER_ID = ?", where `?` will be `setString` with `var3` (as shown in line 4).

5.3.3 Implementation of *Priv*

Priv is implemented as a research prototype and then integrated with a commercial SAST technique (AppScan Source). The commercial SAST technique scans the source code, and stores the detection results (i.e., vulnerability warnings) in an assessment file, which is then displayed by the GUI component of the SAST technique so that users/developers can investigate the detected vulnerability warnings in detail. The assessment file contains comprehensive information of each vulnerability warning, such as trace nodes and trace structure (i.e., a graph structure of trace nodes). The intermediate information, such as code fragments (i.e., *context*) of the detected warnings, is also stored. The *context* information is leveraged by *Priv* to reverse-engineer the vulnerable code and to generate fix suggestions. *Priv* is implemented in a way that it only uses the assessment files, and does not need the project source code.

The implementation of *Priv* consists of three parts. The first component (implemented in Java) reads and analyzes the assessment files, for identifying both preferred fix locations and actionable database- and attribute-related warnings. Then the analysis results are re-directed to the second component that is built using the *D3.js* framework for visualization. For constructing customized remediation pages, *Priv* intercepts the current remediation pages of the underlying SAST technique, then replaces the general buggy and fix code with customized ones for seamless integration. *Priv* should be easily extended to analyze the results of other static-analysis security testing tools, because the information used in *Priv*, such as trace, nodes in the trace, sinks and sources, and vulnerability types, are standard information that is commonly used by other static-analysis security testing techniques.

Table 5.2: The summary of the six projects that are used in the evaluation.

Project	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
LOC (Java)	3,724	321,038	2,645	3,022	1,516	4,176
All the Reported Warnings	281	2,215	159	102	2,424	151
Cross-site Scriptings	43	239	31	14	2,083	9
SQL Injections	55	78	5	12	94	16
Command Injection	0	7	0	2	0	0
Path Traversal	3	60	0	3	18	4
Second-order Injections	0	199	2	0	8	0
Cryptography Insecurity	0	2	0	1	1	0

5.4 Evaluation

In this evaluation, *Priv* is applied to six web applications. All the evaluated projects except for one (AltoroJ) are open-source projects. Table 5.2 summarizes the detailed information of the six web applications: AtoroJ is used for internal evaluation by AppScan Source, and the other five evaluated projects are open-source projects that are commonly used for studying vulnerabilities in web applications. For each evaluated project, AppScan Source is used as the commercial SAST tool to detect vulnerabilities. Then *Priv* is employed as a new remediation system that provides QA resource prioritization via visualization and customized fix suggestions. In summary, the following three research questions are answered:

RQ1: *How do the suggested fix locations by* Priv *compare to the ones that are identified by developers?*

We compared the fix locations suggested by *Priv* with the ones annotated by developers for evaluating whether *Priv* can provide accurate fix location suggestions.

RQ2: *How many actionable warnings can* Priv *find for better diagnosis?*

Priv identifies actionable database- and attribute-related vulnerability warnings by connecting relevant warnings. We evaluate the actual number and the percentage of actionable warnings that can be identified by *Priv*.

RQ3: *What is the quality of the* Priv*'s automatically-generated fixes?*

Priv uses a set of fix templates to generate a customized remediation information *for each detected vulnerability warning*. Thus, whether *Priv* can provide complete and

accurate customized remediation information is evaluated through manual examination of the generated customized remediation information.

RQ1: How do the suggested fix locations by *Priv* compare to the ones that are identified by developers?

Motivation. *Priv* prioritizes the workloads for fixing vulnerability warnings based on shared preferred fix locations. However, finding fix location may be subjective. More specifically, when developers are fixing the detected vulnerabilities without any guidance (e.g., *Priv*), they may choose different locations to add the fixes. For example, developers could choose a different fix location based on their own experience, such as how a similar warning in the same file was fixed in the past. Therefore, a comparison is conducted between the *pFixLocs* chosen by *Priv* and the ones annotated by developers. Note that this comparison shall not be positioned as whether *Priv* suggests better fix locations than developers, or vice versa, but rather a detailed comparison to highlight the similarities and differences between *Priv* and developers.

Approach. As described in details in Section 5.3.1. *Priv* is applied to four types of vulnerabilities: cross-site scriptings, SQL injections, command injections, and path traversals from the six projects listed in Table 5.2. Then, I compare the *pFixLocs* made by *Priv* with the ones annotated by the collaborators from industry who are experienced professional security developer.

Before listing the comparison results, some statistics on the *pFixLocs* provided by *Priv* are provided. First, **reduction** is used to measure to what percentage that *Priv* can narrow down the scope of fix locations. For example, the trace of one vulnerability warning contains N trace nodes (i.e., each trace node is a potential fix location where the validation code can be added). If *Priv* suggests one *pFixLoc*, then the reduction of *Priv* for this vulnerability warning is `(N-1)/N`. Second, we use **cost** to measure the extra cost by *Priv* since *Priv* could suggest more than one fix locations for each vulnerability warning. *Cost* is the number of unique *pFixLocs* suggested by *Priv*.

To show the comparison results on the fix locations, *similarity* is used to measure what percentage of warnings *Priv* suggests identicial fix locations to the ones by annotated developers. Since *Priv* may suggest more than one fix location for one warning, we determines that *Priv* suggests identical fix location if any of the suggested *pFixLocs* matches with the one annotated by the developers.

To conduct the comparison, the annotations of fix location are obtained from experienced security developers. However, conducting a complete one-to-one comparison is not feasible

because of inconsistent annotation styles. More specifically, the annotations are not dedicated for the evaluation of *Priv*, and were used for internal testing purposes. First, when the developers worked on annotating fix locations, they may not annotate the warnings that are resulted from the same SAST configuration setting. Also, the developers may not identify fix locations for every vulnerability warning. Moreover, the underlying SAST product does not contain a consistent identifier to annotate each warning; therefore, when the developers annotated the suggested fix locations, they used the line number of the annotated fix locations to distinguish between different warnings. However, it causes ambiguity since there may exist more than one warning with the same root. I tried my best to resolve the above-mentioned issues through discussions with the developer who provides the annotations. But still, there exist mismatches between the warnings in the evaluation and the ones annotated by the developers. Such mismatches are excluded in this evaluation.

Results. Table 5.3 shows the comparison results between the fix locations suggested by *Priv* and the ones annotated by developers. Due to the reason of inconsistent annotation style mentioned in the *approach* section above, the vulnerability warnings included in this table are a subset of all the warnings from the evaluated projects. Table 5.3 lists the number of warnings with developers' annotated fix locations, and the comparison results with *Priv*'s suggested fix location (i.e., 'cannot find', 'diff', and 'identical'). 'Cannot find' refers to the subset of warnings for which I am not able to find for this experiment due to the inconsistent annotation style, and thus were excluded from calculating the *similarity*. 'Diff' means that the fix locations suggested by *Priv* are different from that of developers. 'Identical' means *Priv* suggests the same fix location as developers. *In summary,* Priv *achieves a 50–100% similarity when comparing the suggested fix locations with the ones annotated by developers.*

Table 5.4 shows the ability of *Priv* in limiting the scope of the manual investigation. For example, for AltoroJ, there are 95 warnings that are either XSS, SQLi, COMMi, or PATHtrv. For these 95 vulnerability warnings, there are a total of 570 possible fix locations (i.e., aggregating all data-flow nodes of vulnerability warnings). *Priv* narrows down the scope to 152 fix locations. Thus the *reduction* of *Priv* on AltoroJ is 73.3%, calculated from (570-152)/570. Moreover, the table also shows the **AVG** cost for each warning that *Priv* analyzed, which means on average how many fix locations were suggested by *Priv*.

RQ2: How many actionable warnings can *Priv* find for better diagnosis?

Motivation. *Priv* identifies actionable database- and attribute-related warnings by connecting two sets of warnings in order to form a complete information-flow: from writing

Table 5.3: The results of comparing fix locations suggested by *Priv* with the ones annotated by developers. Note that the developer did not annotate the fix locations for PATHtrv and COMMi, thus these two types of vulnerability warnings are not shown in this table.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
XSS						
w/ annotations	15	37	69	2	56	15
cannot find	2	15	2	0	17	7
diff	0	11	29	1	0	0
identical	13	11	38	1	17	7
similarity	13/13 (100%)	11/22 (50%)	38/67 (56.7%)	1/2 (50%)	17/17 (100%)	7/7 (100%)
SQLi						
w/ annotations	7	13	6	11	20	16
cannot find	0	1	0	9	1	0
diff	0	1	0	2	3	0
identical	7	11	6	0	16	16
similarity	7/7 (100%)	11/12 (91.7%)	6/6 (100%)	0	16/19(84.2%)	16/16 (100%)

Table 5.4: The table shows the statistics on the preferred fix locations that are identified by *Priv*. *Reduction* shows that *Priv* suggests a few *pFixLocs* from all possible fix locations (i.e., all trace anodes). *AVG cost* shows the average number of fix locations suggested by *Priv* for each vulnerability warning.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
num. of warnings analyzed by *Priv*	95	318	25	27	2,192	29
num. of fix locations suggested by *Priv*	152	429	25	83	2,294	32
num. of trace nodes	570	1,753	129	230	10,923	168
reduction	73.3%	65.5%	80.6%	63.9%	79%	81%
AVG *cost*	1.6	1.34	1	3.07	1.04	1.10

malicious input to a database or attributes, to reading the previously-written malicious input. Identifying actionable warnings benefits developers for better diagnosing and implementing the fixes. This RQ answers how many actionable warnings are identified by *Priv*.

Approach. *Priv* is applied on the six projects in Table 5.2 to identify actionable warnings. For database-related warnings, *Priv* requires manual effort to specify the names of the database tables. For the attribute-related warnings, *Priv* first tries to extract the values of the attributes if the parameters of `getAttribute` or `setAttribute` are string literals. If not, *Priv* then requires manual input to specify the names of the attributes. For database-related warnings in this evaluation, I manually provided the database names. For attribute-related warnings in this evaluation, *Priv* automatically extracts correct names of attributes for all cases. Thus, no manual effort is required for attribute-related warnings.

Results. Table 5.5 concludes the results of applying *Priv* to identify actionable database- and attribute-related warnings. For each part (i.e., either database- or attribute-related), Table 5.5 shows the total number of sets: each set of warnings consists of all the warnings that either write to or read from the same database table or attribute. For example, all the vulnerability warnings that write to the database table '`user`' and the warnings that read from '`user`' are connected by *Priv*, thus resulting in the same set. The number of *actionable* sets refers to the sets whose *actionable* **entry** and **exit** sets of vulnerability warnings are both not empty. Finally, the percentage of actionable warnings is the percentage of both of the actionable entry and exit vulnerability warnings.

The results on Bodgeit are used to explain the table in details. There are five sets of database-related warnings in Bodgeit, and the database tables are `comments`, `users`, `products`, `productType`, and `score`. Among the five sets, only one set contains both entry and exit warnings, and they are related to `comments` (as shown in the row of '# of relevant sets', Table 5.5). For the database table `comments`, there are second-order injections that write to `comments` table. There exist vulnerability warnings (e.g., cross-site scriptings) that read from the same database table–`comments`. Thus, *Priv* locates four actionable vulnerability warnings (**entry**: 2, and **exit**: 2) out of 35 database-related warnings. The remaining 31 can be viewed as less harmful at the moment, until those database tables become vulnerable or exposed by code evolution. For example, for database table `users`, there exist second-order injections that write to it. But there is no code that reads from `users` to expose malicious input. Thus, such second-order injections are not actionable at the moment.

In summary, for the six evaluated projects, *Priv* detects a total of 2,565 actionable database-related or attribute-related vulnerability warnings. The percentage of actionable

Table 5.5: The results of applying *Priv* to identify relevant warnings on the six projects. One set of warnings refers to the warnings that are either related to the same database table or the same attribute. Each row of '# of sets' is the union of both entry and exit sets of warnings. Each row of '# of relevant sets' means the intersection of entry and exit sets and neither entry set nor exit set is empty.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
Database						
# of sets	1	10	5	0	5	0
# of entry vuln. warnings	19	199	4	8	72	16
# of exit vuln. warnings	1	99	31	1	56	0
# of actionable sets	1	6	1	0	3	0
# of actionable entry vuln. warnings	13	152	2	0	53	0
# of actionable exit vuln. warnings	1	81	2	0	56	0
perc. of actionable warnings	70.0%	78.2%	11.4%	N/A	85.2%	N/A
Attribute						
# of sets	8	17	2	0	0	9
# of entry vuln. warnings	8	4	1	0	0	21
# of exit vuln. warnings	9	13,320	10	0	0	8
# of actionable sets	2	2	1	0	0	7
# of actionable entry vuln. warnings	2	2	1	0	0	18
# of actionable exit vuln. warnings	2	2,168	5	0	0	7
perc. of actionable warnings	23.5%	16.3%	54.5%	N/A	N/A	86.2%

warnings ranges from 11.4% to 86.2%. Developers could consider fixing the actionable vulnerability warnings first since they are more dangerous than the others.

RQ3: What is the quality of the *Priv*'s automatically-generated fixes?

Motivation. *Priv* aims to provide customized remediation pages that include fix suggestions to elevate the burden from developers. Particularly, *Priv* generates fix suggestions for five vulnerability types: cross-site scriptings, SQL injections, path traversals, command injections, and cryptography insecurities. All of the five except for one are also studied in *Priv*–Phase I because cryptography insecurity vulnerability warnings typically only contain one trace node (i.e., sink), thus there is no need to suggest *pFixLoc* for them and present them in the global-view visualization. The best fixes by *Priv* can be directly applied to the source code (i.e., contains complete essential and correct semantics to fix the detected vulnerability). We want to study how many customized remediation pages (i.e., showing both the buggy code and the corresponding fixes) can *Priv* generate given all the detected

vulnerability warnings. Moreover, we evaluate the quality of the fixes in the remediation pages regarding whether the fix can be directly applied to the source code, and if not, how much additional work is required.

Approach. First, I identified the number of vulnerability warnings for which *Priv* can provide customized remediation pages (i.e., buggy code and the corresponding fix). Second, I manually checked the quality of each suggested fix generated by *Priv*, except for XSS from JavaVulnLab (totally 2,083 XSS), for which a sample of 100 is taken for manual examination. To quantitatively evaluate the quality of the fixes by *Priv*, we manually classify each fix into one of the following three categories: *fully complete*, *partially compilable/correct* (i.e., minor modifications required), *fix template only* (i.e., developers need to add essential code, such as validation methods). The category–*fully complete* refers to the fixes that can be directly applied to the source code without modifications, which means the patched code is compilable and indeed fixes the targeted vulnerability finding. The category–*partially compilable/correct* shows that the suggested fix cannot be directly applied to source code since the patched code would be either not-compilable or not totally correct. The fixes of this category require minor modifications to be compilable and correct. For example, Figure 5.12 shows one SQL injection for which *Priv* generates a fix that is not completely correct. The fix is not entirely correct because, in line 8 and 13, `dateString` should not be replaced by '?' since `dateString` is used in a condition instead of concatenating to a SQL string.

The least complete category–*fix template only* refers to a fix that only contains the skeleton of the fix, but lacks concrete implementation, such as the validation method for checking malicious inputs. For example, to prevent a path traversal in the code `File file = new File(path)` when `path` could be malicious input from users, *Priv* suggests validating `path`. The best practice of the validation method is to create a whitelist that specifies valid paths; therefore *Priv* leaves the implementation of this validation method to developers, and provides a fix template `if(validate(path))`.

Results. Table 5.6 lists the results of manual examination on the quality of the fixes generated by *Priv*. Each fix is manually examined and classified to one of the three categories explained in the approach section (i.e., *fully complete*, *partially compilable/correct*, and *fix template only*).

For all the path traversals and command injections, *Priv* only provides incomplete fixes due to lack of detailed implementation of validation methods (*fix template only*). The concrete validation method requires the knowledge of the context of PATHtrv or COMMi, such as the range of files that can be safely accessed (PATHtrv).

For most of the cross-site scriptings (up to 100%), *Priv* provides complete and correct

```
if (startDate != null && startDate.length()>0 && endDate != null && endDate.length()>0){
   dateString = "DATE␣BETWEEN␣" +startDate +"␣00:00:00␣AND␣" +endDate +"␣23:59:59";
} else if (startDate != null && startDate.length()>0){
   dateString = "DATE␣>␣" + startDate +"␣00:00:00";
}
- String query = "SELECT␣*␣FROM␣TRANSACTIONS␣WHERE␣("
                        + acctIds.toString() + ")␣"
                        + ((dateString==null)?"": "AND␣("
                        + dateString + ")␣")
                        + "ORDER␣BY␣DATE␣DESC" ;
+   String query = "SELECT␣*␣FROM␣TRANSACTIONS␣WHERE␣("
                        + "?" + ")␣"
                        + (("?"==null)?"": "AND␣("
                        + "?"+ ")␣")
                        + "ORDER␣BY␣DATE␣DESC" ;
+   query.setString(1, acctIds.toString());
+   query.setString(2, dateString);
+   query.setString(3, dateString);
   resultSet = statement.executeQuery(query);
```

Figure 5.12: A fix suggestion that is automatically generated by *Priv* for a SQL injection vulnerability. The fix suggestion is classified as 'partially compilable/correct' upon manual examination. +/- represents *Priv*'s suggested code changes (i.e., add or remove the corresponding line of code).

```
<%= (request.getAttribute("message_feedback")!=null)?"...%>.
```

Figure 5.13: An example of cross-site scripting from AltoroJ that *Priv* generates partially compilable/correct fix for.

fixes. For the 21 cross-site scriptings from AltoroJ, *Priv* does not provide fully complete fixes because the assessment file (which *Priv* uses as input) from AppScan Source does not always store the original information of the tainted parameter of the sink. Figure 5.13 shows an example of such cases, `request.getAttribute(...) != null` is stored as `Temp@11@0` in the assessment file. Thus, *Priv* is not able to infer that `request.getAttribute()` is used in a condition, and therefore replacing the function call in an `HTMLEncode` validation method, which is not desired (*partially compilable/correct*). *Priv* generates partially compilable/correct fixes for one vulnerability warning from Bodgeit, and five from JavaVulnLab for the same reason.

For SQL injections, *Priv* generates fully complete fixes for 70.9% (on average) of the detected SQL injections in the evaluated projects. The primary reason that *Priv* generates the fixes of the two incomplete categories (i.e., partially compilable/correct and fix template only) is the same with that for cross-site scriptings. Specifically, for the example shown in

Table 5.6: The Results of Manual Examination on the Qualify of the Fixes by *Priv*.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
Cross-site Scripting						
fully complete	22	239	30	14	95[1]	9
partially compilable/correct	21	0	1	0	5	0
fix template only	0	0	0	0	0	0
SQL Injection						
fully complete	42	78	2	0	51	0
partially compilable/correct	6	0	0	0	0	0
fix template only	7	0	3	12	43	9
Path Traversal						
fully complete	0	0	N/A	0	0	0
partially compilable/correct	0	0	N/A	0	0	0
fix template only	3	60	N/A	3	18	4
Command Injection						
fully complete	N/A	0	N/A	0	N/A	N/A
partially compilable/correct	N/A	0	N/A	0	N/A	N/A
fix template only	N/A	7	N/A	2	N/A	N/A
Cryptography Insecurity						
fully complete	N/A	2	N/A	1	1	N/A
partially compilable/correct	N/A	0	N/A	0	0	N/A
fix template only	N/A	0	N/A	0	0	N/A

Figure 5.12, AppScan Source uses intermediate variables to store code elements such as `dateString==null`. Therefore *Priv* is unable to analyze that condition to avoid producing the incorrect fix: replacing `dateString` with '?'.

5.5 A Case Study on the Performance of Prioritizing Quality-Assurance Effort by *Priv*

This section provides a case study on how *Priv* can prioritize quality-assurance effort. For example, if the number of clusters in the global-view visualization of *Priv* is much smaller than the number of individual warnings, *Priv* can indeed prioritize developers' quality-assurance effort by guiding them to work on the clusters of significant size first. Intuitively,

[1] based on a sample of 100

Table 5.7: The table shows that for AltoroJ, how effective *Priv* is at prioritizing quality-assurance efforts.

# of fix locations investigated	8	12	14	15	16	17	18	19
# of warnings resolved	41	48	52	54	56	58	60	61
AVG warnings resolved per fix location	5.13	4	3.71	3.6	3.5	3.41	3.33	3.21

investigating the largest cluster has a chance to resolve many vulnerability warnings at once (i.e., they share common fix locations). Therefore, to measure the effectiveness of *Priv* in prioritizing workloads, we performed a simulated study on the AltoroJ project. Particularly, with the assumption that developers work on the biggest cluster of vulnerability warnings, we sort the clusters based on the number of vulnerability warnings involved. Then, we record the effort of fixing the current biggest cluster (i.e., number of fix locations suggested) and the outcome of the fixing effort (i.e., the number of warnings in this cluster).

Table 5.7 shows how *Priv* can prioritize quality-assurance efforts for AltoroJ (one of the evaluated projects). The order of each row complies with the order that developers follow based on the assumption (i.e., starting with the most significant cluster). The first row shows the aggregated number of fix locations that developers would need to investigate. Correspondingly, the second row lists the number of fixed warnings resolved if following each choice of a cluster. The last row shows the average number of warnings that would be resolved per fix location. For example, when developers choose to work on the first cluster, 41 vulnerability warnings would be resolved if investigating eight fix locations. If there is no prioritization, for each fix location, developers would resolve 3.21 warnings on average (as shown in the last column in Table 5.7). With the prioritization by *Priv*, the average warnings per fix location are 5.13 to start with, and gradually decreases to 3.21. This shows that *Priv* can indeed improve work efficiency by prioritizing quality-assurance efforts.

5.6 Threats to Validity

Priv is designed and built on one commercial product–AppScan Source. Although I believe *Priv* can be integrated with other SAST techniques, it is not evaluated how effective that would be, which remains as future work. The fix location suggestions by *Priv* are compared with with the ones by developers (*RQ1*) based on the annotations from one security expert. For each evaluated project, the security expert did not leave annotations of fix locations for every detected vulnerability warning. Thus, for *RQ1*, I was able to conduct the comparison

for a subset of all the warnings. Future research includes a user study to obtain such fix location annotations for all the applicable warnings of each project from developers.

5.7 Conclusions

This chapter proposes *Priv*, which is a multi-phase remediation system to assist developers in fixing vulnerability warnings by SAST techniques more efficiently. *Priv* prioritizes developers' quality-assurance efforts by visualizing the clusters of vulnerability warnings based on shared preferred fix locations. Moreover, *Priv* provides essential information that could boost the diagnosis and fixing process of vulnerability warnings. Specifically, *Priv* identifies actionable warnings by locating relevant warnings for database- and attribute-related warnings. Finally, *Priv* improves current remediation pages with customized remediation that includes buggy code and fix code. The evaluation on six web applications shows that *Priv* suggests identical fix locations to the ones by developers for many warnings. Also, *Priv* prioritizes developer's workloads by making developers aware of the commonality of vulnerability warnings regarding fix locations. We also show how many actionable database-related and attribute-related warnings are identified by *Priv*. Finally, the quality of the generated fixes from the customized remediation pages are categorized into four categories of completeness. *Priv* is able to provide complete fixes to many of the warnings.

Chapter 6

Conclusions and Future Work

6.1 Thesis Conclusions

This thesis improves the correctness of current automated program repair: limited correctness guarantee of the automatically-generated patches, and limited hypothesized space of high-quality patches. One one hand, this thesis proposes *Opad* to improve the test-suite-based validation in G&V techniques. The findings on the automatically-generated test cases highlight that weak oracles prevent the further improvement of G&V techniques. On the other hand, for enriching the hypothesized space of high-quality patches, this thesis leverages human knowledge for improving automated program repair: *APARE* and *Priv*. *APARE* targets project-specific recurring bugs and utilizes past fixes that are considered as high-quality patches crafted by human beings. *Priv* targets vulnerability warnings by static application security testing, and provides workload prioritization, and fix suggestions based on rules that are developed based on security expertise.

6.2 Future Research Directions

Use of Overfitted Patches. It is commonly believed that the overfitted patches from G&V techniques, which are incorrect, are not useful. However, I believe the overfitted patches, which pass regression test suites, contain important information for debugging and bug fixing. Hence, in the future, I plan to explore the use of the overfitted patches. First, I plan to conduct a user study on whether the overfitted patches can help developers generate correct patches. Second, I plan to explore whether overfitted patches can be used

to guide G&V techniques generating correct patches. For example, once overfitted patches are identified, the identified overfitted patches can be used to identify other overfitted patches in the hypothesized space of patches, such as based on similarity of execution traces or to rank the patches in the space.

Detect and Repair API Misuses in Data Analytics Code by Leveraging Data-Sensitive Fix Patterns. More and more applications leverage open-source frameworks and libraries for data analytics (e.g., Spark, Hadoop, TensorFlow, scikit-learn, weka, etc.). Despite the differences of such applications, the code segments that use the same API of the data analytics library or framework typically have similar logic and code structure. Previous studies on API misuse detection do not automatically fix the misuses. Moreover, such data analytics API misuses and their fixes are data sensitive, which should be taken into consideration. I plan to propose a technique that extracts bug patterns and corresponding fix patterns of API misuses in data analytics code by mining large-scale code repositories (e.g., GitHub), and applies the learnt patterns to detect and repair previously-unknown bugs.

A Recommendation Technique for Automated Program Repair. In recent years, more than a dozen APR techniques have been proposed. The proposed APR techniques complement each other in fixing bugs [33], instead of existing one winner APR technique that beats the rest. Given a bug to repair, to fully explore whether APR techniques can repair the bug, one needs to apply every APR technique to repair the bug, which is time-consuming and expensive. Also, developers need to spend time and effort to identify the incorrect patches by APR techniques and choose the correct patches. To improve the efficiency of applying APR techniques, I plan to propose a recommendation tool for choosing one or a few APR techniques for a given bug. I anticipate that the proposed technique can improve the efficiency of APR techniques by saving time and reducing incorrect patches. For example, if one APR technique is not suitable for a given bug, then it should not be chosen to repair the bug. The proposed recommendation technique can leverage a collection of bugs, which have been evaluated by current APR techniques (i.e., the training set), for recommending a given new bug, which APR techniques would be suitable. I plan to explore features for classification, such as the characteristics of test suites (e.g., coverage metrics), and the characteristics of the top-ranked faulty locations (e.g., types of statements). In addition, I plan to inspect the features that can represent the quality of test suite used for validation. I believe that different APR techniques have different specialty: one may be more tolerant of low-quality test suites, or one may be better at repairing conditions.

Cross-Component Automated Program Repair. Current APR techniques focus on repairing bugs by modifying one component of the applications, i.e., source code files. However, modern applications typically contains more than one component, e.g., source

code, test cases, configurations, and build scripts. Bugs that require fixes across multiple components are not handled well by current techniques. I plan to propose new APR techniques that target the bugs that require fixes spanned across multiple components. I plan to leverage both static and dynamic analysis to build the cross-component links, such as between configurations and source code [146, 151], and across configurations, source code, and test cases [27]. The inferred links can be used for extracting cross-component fix patterns and guiding the search of the correct cross-component repair.

Human-in-the-Loop Repair System. Human knowledge has great potential to improve automated program repair. The existing research typically incorporates human knowledge (e.g., past fixes or fix patterns) at the early stage such as when researchers design the techniques [75, 82, 134], or at the post-mortem stage such as utilized by a ranking strategy on the automatically-generated patches by APR [68].

In the future, I plan to incorporate in-depth human knowledge with APR techniques based on the human-in-the-loop methodology. Similar to human-in-the-loop for machine learning, for automated program repair, I also believe that machines and human knowledge are winning combination. On one hand, machines can be faster at reasoning than human by automation (e.g., validation by test cases), and more diligent at edge cases through iteration. On the other hand, human beings (i.e., developers) have domain expertise that cannot be fully encoded, and thus can be automated by machines. For example, the current APR techniques rely on test cases to infer partial specifications [94, 83]. However, complete specifications (i.e., all correct and incorrect incorrect behaviors) of software are tremendously challenging to obtain. Therefore, automated program repair can benefit significantly from the input from developers which specifies correct/incorrect behaviors. The future plan is to propose an interactive repair system that enables human-in-the-loop methodology. To begin with, I plan to add live judgments from developers in deciding the correctness of the automatically-generated patches. I plan to design the repair system to consider the feedback from developers when generating the next patch. Furthermore, I plan to design the repair system to take input (e.g., correct/incorrect behaviors) from developers on the fly (i.e., before generating a patch). It remains as future work to explore when and what to provide developers for balancing overhead and efficiency.

References

[1] American fuzzy lop, 2017. http://lcamtuf.coredump.cx/afl/.

[2] Checkmark static code analysis, 2017. https://www.checkmarx.com/technology/static-code-analysis-sca.

[3] Fortify static code analyzer, 2017. https://software.microfocus.com/it-it/software/sca.

[4] gcov—a test coverage program, 2017. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[5] tiffcp manual page, 2017. http://www.remotesensing.org/libtiff/man/tiffcp.1.html.

[6] Valgrind, 2017. http://valgrind.org/.

[7] zzuf: multi-purpose fuzzer, 2017. http://caca.zoy.org/wiki/zzuf.

[8] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, November 2009.

[9] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Viktor Kunčak. Repairing decision-making programs under uncertainty. In *Computer Aided Verification*, pages 181–200, 2017.

[10] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA, pages 225–236, 2014.

[11] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Proceedings of the 2013 Formal Methods in Computer-Aided Design*, CAV, pages 1–8, 2013.

[12] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[13] 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*, Eclipse, pages 35–39, 2005.

[14] John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10:1–10:35, August 2011.

[15] A. Arcuri, M.Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, March 2012.

[16] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 259–269, 2014.

[17] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA, pages 805–806, 2007.

[18] 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*, FSE, pages 306–317, 2004.

[19] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA, pages 257–269, 2015.

[20] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE, pages 625–634, 2014.

[21] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, pages 308–318, 2006.

[22] Sam Blackshear and Shuvendu Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 209–218, 2013.

[23] 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*, ESEC/FSE, pages 117–128, 2017.

[24] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 209–224, 2008.

[25] 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*, ASE, pages 637–647, 2017.

[26] T. H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Detecting problems in the database access code of large scale systems - an industrial experience report. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, ICSE-SEIP, pages 71–80, 2016.

[27] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 666–677, 2016.

[28] Zack Coker and Munawar Hafiz. Program transformations to fix c integers. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pages 792–801, 2013.

[29] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, September 2004.

[30] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: Synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 582–592, 2017.

[31] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *Computer Aided Verification*, volume 9780 of *CAV*, pages 383–401, 2016.

[32] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA, pages 30–39, 2014.

[33] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, Aug 2015.

[34] Ansgar Fehnker, Ralf Huuck, Sean Seefried, and Michael Tapp. Fade to grey: Tuning static program analysis. *Electronic Notes in Theorectical Computer Science*, 266:17–32, October 2010.

[35] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[36] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE, pages 416–419, 2011.

[37] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Extending a search-based test generator with adaptive dynamic symbolic execution (tool paper). In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA, pages 421–424, 2014.

[38] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for c programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1 of *ICSE*, pages 459–470, 2015.

[39] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing q&a sites (t). In *Proceedings of the 30th IEEE/ACM*

International Conference on Automated Software Engineering, ASE, pages 307–318, 2015.

[40] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[41] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI Conference on Artificial Intelligence*, pages 1–7, 2017.

[42] William G. J. Halfond and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 174–183, 2005.

[43] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI, pages 69–82.

[44] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, (4):279–290, 1977.

[45] 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*, MSR, pages 152–161, 2014.

[46] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on Software engineering*, ICSE, pages 274–283, 2005.

[47] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pages 392–401, 2013.

[48] IBM. Appscan source, 2017. https://www.ibm.com/us-en/marketplace/ibm-appscan-source.

[49] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE, pages 435–445, 2014.

[50] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE, pages 96–105, 2007.

[51] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 221–236, 2012.

[52] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE, pages 474–484, 2012.

[53] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pages 672–681, 2013.

[54] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP, pages 258–263, 2006.

[55] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *Proceedings of the 12th International Conference on Static Analysis*, SAS, pages 203–217, 2005.

[56] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. Smt-based false positive elimination in static program analysis. In *Proceedings of the 14th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM, pages 316–331, 2012.

[57] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 654–665, 2004.

[58] 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*, ICSE, pages 266–276, 2014.

[59] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654 – 670, July 2002.

[60] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 295–306, 2015.

[61] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pages 802–811, 2013.

[62] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE, pages 309–319, 2009.

[63] Miryung Kim, Vibha Sazawal, and David Notkin. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, FSE, pages 187–196, 2005.

[64] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS, pages 40–56, 2001.

[65] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT/FSE, pages 83–93, 2004.

[66] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE, pages 492–501, 2006.

[67] Wei Le and Mary Lou Soffa. Path-based fault correlations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 307–316, 2010.

[68] X. B. D. Le, D. Lo, and C. L. Goues. History driven program repair. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1 of *SANER*, pages 213–224, 2016.

[69] 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 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 593–604, 2017.

[70] 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 2012 International Conference on Software Engineering*, ICSE, pages 3–13, 2012.

[71] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176 – 192, March 2006.

[72] Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Inferring project-specific bug patterns for detecting sibling bugs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 565–575, 2013.

[73] Yiyan Lin and Sandeep S. Kulkarni. Automatic repair for multi-threaded program with deadlock/livelock using maximum satisfiability. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA, pages 237–247, 2014.

[74] Richard J Lipton, Richard A DeMillo, and FG Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[75] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST, pages 282–291, 2013.

[76] Haopeng Liu, Yuxi Chen, and Shan Lu. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 715–726, 2016.

[77] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE, pages 318–329, 2014.

[78] Xinyuan Liu, Muhan Zeng, Yingfei Xiong, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based automatic program repair. *CoRR*, abs/1706.09120, 2017.

[79] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, SSYM, pages 18–18, 2005.

[80] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. *SIGPLAN Not.*, 47(10):133–146, October 2012.

[81] Francesco Logozzo and Matthieu Martel. Automatic repair of overflowing expressions with abstract interpretation. In *CoRR*, volume abs/1309.5148, 2013.

[82] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 727–739, 2017.

[83] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *In proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering*, FSE, pages 166–178, 2015.

[84] 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*, ICSE, pages 702–713, 2016.

[85] 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*, POPL, pages 298–312, 2016.

[86] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS, pages 229–240, 2012.

[87] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. Automated repair of layout cross browser issues using search-based techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 249–260, New York, NY, USA, 2017. ACM.

[88] Paul Dan Marinescu and Cristian Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE, pages 716–726, 2012.

[89] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM, pages 388–391, 2013.

[90] 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, February 2015.

[91] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Journal of Empirical Software Engineering*, 20(1):176–205, February 2015.

[92] 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 *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE-C, pages 492–495, 2014.

[93] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DTIC Document, 2012.

[94] 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*, ICSE, pages 691–701, 2016.

[95] Na Meng, Miryung Kim, and Kathryn S. McKinley. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE, pages 440–443, 2011.

[96] 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*, ICSE, pages 502–511, 2013.

[97] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[98] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[99] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 2000 International Conference on Software Maintenance*, ICSM, pages 120–120, 2000.

[100] 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*, ICSE, pages 234–242, 2014.

[101] Martin Monperrus. Automatic Software Repair: a Bibliography. *ACM Computing Surveys*, 2017.

[102] Paul Muntean, Vasantha Kommanapalli, Andreas Ibing, and Claudia Eckert. Automated generation of buffer overflow quick fixes using symbolic execution and smt. In *Proceedings of the 34th International Conference on Computer Safety, Reliability, and Security - Volume 9337*, SAFECOMP, pages 441–456, 2015.

[103] T. B. Muske, A. Baid, and T. Sanas. Review efforts reduction by partitioning of static analysis warnings. In *Proceedings of the 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation*, SCAM, pages 106–115, 2013.

[104] Stats Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE, pages 803–813, 2014.

[105] 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 International Conference on Automated Software Engineering*, ASE, pages 180–190, 2013.

[106] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 302–321, 2010.

[107] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pages 772–781, 2013.

[108] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE, pages 315–324, 2010.

[109] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM - Surviving the data deluge*, 51(12):87–95, December 2008.

[110] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[111] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[112] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE, pages 75–84, 2007.

[113] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, June 2009.

[114] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. An empirical study of supplementary bug fixes. In *Proceedings of the 2012 9th IEEE Working Conference on Mining Software Repositories*, MSR, pages 40–49, 2012.

[115] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP, pages 87–102, 2009.

[116] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE, pages 254–265, 2014.

[117] 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*, ISSTA, pages 24–36, 2015.

[118] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. *Counterexample-Guided Quantifier Instantiation for Synthesis in SMT*, pages 198–216. 2015.

[119] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering*, ICSE, pages 404–415, 2017.

[120] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE, pages 341–350, 2008.

[121] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2006.

[122] Address Sanitizer. Address sanitizer, 2016. https://github.com/google/sanitizers.

[123] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 263–272, 2005.

[124] Jooyong Yi Sergey Mechtaev and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE, pages 448–458, 2015.

[125] H. Shen, J. Fang, and J. Zhao. Efindbugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 299–308, 2011.

[126] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 43–54, 2015.

[127] 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*, ESEC/FSE, pages 532–543, 2015.

[128] Y. Song, X. Wang, T. Xie, L. Zhang, and H. Mei. Jdf: detecting duplicate bug reports in jazz. In *Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 315–316, 2010.

[129] M. Surf and A. Shulman. How safe is it out there? zeroing in on the vulnerabilities of application security. In *Imperva Application Defense Center Paper*, 2004.

[130] Michael Sutton. Filefuzz, 2017. http://www.securiteam.com/tools/5PP051FGUE.html.

[131] Roberto Tamassia. *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. 2007.

[132] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST, pages 260–269, 2012.

[133] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *Proceedings of the 2015 Internaltional Conference on Software Engineering*, ICSE, pages 471–482, 2015.

[134] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 727–738, 2016.

[135] 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 the Foundations of Software Engineering*, FSE, pages 64–74, 2014.

[136] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 752–762, 2017.

[137] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 762–774, 2014.

[138] OWASP WebGoat. https://github.com/webgoat, 2017.

[139] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 356–366, 2013.

[140] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: Interactive system configuration repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 625–636, 2017.

[141] 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*, ASE, pages 87–98, 2016.

[142] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP, pages 380–403, 2006.

[143] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 226–236, 2017.

[144] 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*, ASE, pages 660–670, 2017.

[145] 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*, ICSE, pages 416–426, 2017.

[146] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP, pages 244–259, 2013.

[147] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[148] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 831–841, 2017.

[149] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[150] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. *ArXiv e-prints*, March 2017.

[151] Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE, pages 152–163, 2014.

[152] Tianyi Zhang, Myoungkyu Song, Joseph Pinebo, and Miryung Kim. Interactive code review for systematic changes. In *Proceedings of the 2015 Internaltional Conference on Software Engineering*, ICSE, pages 111–122, 2015.

[153] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 214–224, 2015.

[154] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 2015 Internaltional Conference on Software Engineering*, ICSE, pages 913–923, 2015.

[155] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572, 2004.

[156] Davor Čubranić and Gail Murphy. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, pages 92–97, 2004.