Machine Learning for Software Dependability

by

Thibaud Lutellier

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, 2020

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:	Martin Monperrus Professor, Division of Theoretical Computer Science, KTH Royal Institute of Technology
Supervisor:	Lin Tan Professor, Electrical & Computer Engineering, University of Waterloo
Internal Member:	Vijay Ganesh Professor, Electrical & Computer Engineering, University of Waterloo
	Derek Rayside Professor, Electrical & Computer Engineering, University of Waterloo
Internal-External Member:	Michael Godfrey Professor, Dept. of Computer Science, University of Waterloo

Author's Declaration

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

As lead author of all the contributions of this thesis, I was responsible for contributing to conceptualizing study design, developing, implementing, and evaluating prototypes and evaluating them, carrying out data collection and analysis, drafting and submitting manuscripts, and presenting the work to conferences. My coauthors provided feedback during each step of the research and on draft manuscripts.

Research presented in Chapter 2: Lawrence Pang helped implementing the context-aware model, Hung Viet Pham and Yitong Li helped with evaluating previous automatic program repair tools for C and C++. Moshi Wei helped in gathering training data. Dr. Lin Tan provided feedback and supervision in all parts of the research.

Research presented in Chapter 3: Dr. Tomasz Kuchta helped to design and collect the regular expressions for the clustering. Dr. Edmund Wong and Dr. Tomasz Kuchta helped manually checking PDF files for inconsistencies. Dr. Christian Cadar and Dr. Lin Tan provided feedback and supervision in all parts of the research.

Research presented in Chapter 4: Devin Chollak helped gathering the dependencies and ground truth of Java projects. Dr. Joshua Garcia provided access and support with the Arcade framework that implements most architecture recovery techniques. Dr. Robert Kroeger helped validating the ground truth of Chromium. Dr. Lin Tan, Dr. Nenad Medvidovic, and Dr. Derek Rayside provided feedback and supervision in all parts of the research.

Citations:

- Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan, *CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair*, Proceeding of The ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020 [157].
- Tomasz Kuchta*, Thibaud Lutellier*, Edmund Wong, Lin Tan, and Cristian Cadar, *On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files*, (* The first two authors contributed equally to this paper), Empirical Software Engineering, 2018 [118].
- Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger, *Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques*, IEEE Transactions on Software Engineering, 2017 [156].

- Thibaud Lutellier, Devin Chollack, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic and Robert Kroeger, *Comparing Software Architecture Recovery Techniques Using Accurate Dependencies,* Proceeding of ACM/IEEE International Conference on Software Engineering, SEIP track, 2015 [154].

Abstract

Dependability is an important quality of modern software but is challenging to achieve. Many software dependability techniques have been proposed to help developers improve software reliability and dependability such as defect prediction [83, 96, 249], bug detection [6, 17, 146], program repair [51, 127, 150, 209, 261, 263], test case prioritization [152, 250], or software architecture recovery [13, 42, 67, 111, 164, 240].

In this thesis we consider how machine learning (ML) and deep-learning (DL) can be used to enhanced software dependability through three examples in three different domains: automatic program repair, bug detection in electronic document readers, and software architecture recovery.

In the first work we propose a new G&V technique—CoCoNuT, which uses ensemble learning on the combination of convolutional neural networks (CNNs) and a new context-aware neural machine translation (NMT) architecture to automatically fix bugs in multiple programming languages. To better represent the context of a bug, we introduce a new context-aware NMT architecture that represents the buggy source code and its surrounding context separately. CoCoNuT uses CNNs instead of recurrent neural networks (RNNs), since CNN layers can be stacked to extract hierarchical features and better model source code at different granularity levels (e.g., statements and functions). In addition, CoCoNuT takes advantage of the randomness in hyperparameter tuning to build multiple models that fix different bugs and combines these models using ensemble learning to fix more bugs. CoCoNuT fixes 493 bugs, including 307 bugs that are fixed by none of the 27 techniques with which we compare.

In the second work, we present a study on the correctness of PDF documents and readers and propose an approach to detect and localize the source of such inconsistencies automatically. We evaluate our automatic approach on a large corpus of over 230K documents using 11 popular readers and our experiments have detected 30 unique bugs in these readers and files.

In the third work we compare software architecture recovery techniques to understand their effectiveness and applicability. Specifically, we study the impact of leveraging accurate symbol dependencies on the accuracy of architecture recovery techniques. In addition, we evaluate other factors of the input dependencies such as the level of granularity and the dynamic-bindings graph construction. The results of our evaluation of nine architecture recovery techniques and their variants suggest that (1) using accurate symbol dependencies has a major influence on recovery quality, and (2) more accurate recovery techniques are needed. Our results show that some of the studied architecture recovery techniques scale to very large systems, whereas others do not.

Acknowledgements

I would like to particularly thank my supervisor, Dr. Lin Tan: thank you for taking me as a student, helping and supporting me during all these years. I learned and grew so much under her supervision and mentorship, I could not have wished for a better supervisor!

I would like to thank my examination committee members—Dr. Martin Monperrus, Dr. Thomas Zimmermann, Dr. Sebastian Fischmeister, and Dr. Vijay Ganesh, Dr. Derek Rayside, and Dr. Michael Godfrey for their support, valuable feedback, and insightful discussion on my research.

Thanks to all the students in our research team at the University of Waterloo who made going to the lab every day a pleasure: Dr. Jaechang Nam, Dr. Jinqiu Yang, Dr. Edmund Wong, Dr. Song Wang, Dr. Nasir Ali, Lei Zhang, Quinn Hanam, Sandeep Chaudhary, Devin Chollak, Ming Tan, Michael Chong, Taiyue Liu, Yuefei Liu, Alexey Zhikhartsev, Yuan Xi, Moshi Wei, Hung Pham, and Yitong Li—thank you very much for all the friendship, support, and joy.

I also want to thank the team at Purdue University, while the opportunities to meet in person were rare, we had great online conversations and I learned a lot from them: Nan Jiang, Shangshu Qian, Jonathan Rosenthal, Jiannan Wang, Danning Xie, and Dr. Mijung Kim.

I want to thank all my other collaborators during my Ph.D. study: Dr. Joshua Garcia, Dr. Nenad Medvidovic, Dr. Robert Kroeger, Dr. Tomasz Kuchta, Dr. Cristian Cadar, Lawrence Pang, Dr. Yaoliang Yu, Dr. Nachiappan Nagappan, Weizhen Qi, Dr. Hossain Shahriar, Dr. Komminist Weldemariam, and Dr. Mohammad Zulkernine. I really enjoyed working with them, and have learned a lot from them.

Dedication

This is dedicated to Sarah. You supported me through all these years, this work is also here thanks to you.

Table of Contents

List of Figures xiv

List of Tables xvi

1 Introduction 1

1.1 Project 1 – Combining Context-Aware Neural Translation Models using Ensemble for Automated Program Repair: . . . 2

1.2 Project 2 – On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files: . . . 2

1.3 Project 3 – Measuring the Impact of Dependencies on Software Architecture Recovery Techniques: . . . 3

1.4 Publications . . . 4

2 Related Work 6

2.1 Neural Machine Translation Models for APR . . . 6

2.1.1 Deep Learning for Automatic Program Repair: . . . 6

2.1.2 G&V Program Repair: . . . 7

2.1.3 Grammatical Error Correction (GEC): . . . 7

2.1.4 Deep Learning in Software Engineering: . . . 7

2.2 Finding, and Localizing Inconsistency Bugs in PDF Readers and Files . . . 7

2.2.1 Cross PDF Reader Inconsistency: . . . 7

2.2.2 Cross Browser Inconsistency: . . . 8

2.2.3 Vulnerabilities in PDF files: . . . 9
2.2.4 Differential Testing: . . . 10
2.2.5 Document Recovery: . . . 10
2.3 Software Architecture Recovery Techniques . . . 11
2.3.1 Comparison of Software Architecture Recovery Techniques: . . . 11
2.3.2 Recovery of Ground-Truth Architectures: . . . 12

3 CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair 14
3.1 Motivation . . . 14
3.2 Background and terminology . . . 18
3.3 Approach . . . 18
3.3.1 Challenges . . . 19
3.3.2 Data Extraction . . . 20
3.3.3 Input Representation and Tokenization . . . 21
3.3.4 Context-Aware NMT Architecture . . . 22
3.3.5 Ensemble Learning . . . 25
3.3.6 Patch Validation . . . 25
3.3.7 Generalization to Other Languages . . . 26
3.4 Experimental Setup . . . 26
3.5 Evaluation and Results . . . 28
3.5.1 RQ1: How does CoCoNuT perform against state-of-the-art APR techniques? . . . 29
3.5.2 RQ2: Which bugs only CoCoNuT can fix? . . . 33
3.5.3 RQ3: What are the contributions of the different components of CoCoNuT and how does it compare to other NMT-based APR techniques? 35
3.5.4 RQ4: Can we explain why CoCoNuT can (or fail to) generate specific fixes? . . . 36
3.5.5 Execution Time . . . 39
3.6 Threats to Validity . . . 40
3.7 Summary . . . 40

4 On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files 41

4.1 Motivation . . . 41
4.2 Approach . . . 44
4.2.1 Phase 1: Document filtering . . . 46
4.2.2 Phase 2: Inconsistency detection . . . 47
4.2.3 Phase 3: Inconsistency localization . . . 49
4.3 Experimental setup . . . 51
4.3.1 Data set . . . 51
4.3.2 Portfolio of readers . . . 54
4.3.3 Infrastructure . . . 55
4.3.4 Research questions. . . . 56
4.4 A Study of Cross-reader Inconsistencies . . . 57
4.4.1 Methods . . . 57
4.4.2 Inconsistencies are common . . . 58
4.4.3 Types of inconsistencies . . . 59
4.5 Automatic results . . . 63
4.5.1 Results for the random sample . . . 63
4.5.2 Results for the entire set . . . 65
4.6 Inconsistency Localization Evaluation . . . 69
4.7 Cross-OS Inconsistencies . . . 70
4.7.1 PDF Readers on Windows . . . 71
4.7.2 Experiment Details . . . 71
4.7.3 Results . . . 71
4.8 Discussion and Threats to Validity . . . 73
4.8.1 Conclusion Validity . . . 73
4.8.2 Internal Validity . . . 73
4.8.3 Construct Validity . . . 74

4.8.4 External Validity . . . 74
4.8.5 Practical Applicability . . . 74
4.9 Summary . . . 75

5 Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques 76

5.1 Motivation . . . 76
5.2 Approach . . . 79
5.2.1 Obtaining Dependencies . . . 80
5.2.2 Obtaining Ground-Truth Architectures . . . 83
5.3 Selected Recovery Techniques . . . 86
5.4 Experimental Setup . . . 88
5.4.1 Projects and Experimental Environment . . . 89
5.4.2 Extracted Dependencies . . . 89
5.4.3 Ground-truth architectures . . . 90
5.4.4 Architecture Recovery Software and Parameters . . . 91
5.4.5 Accuracy Measures . . . 91
5.5 Evaluation and Results . . . 94
5.5.1 RQ1: Can accurate dependencies improve the accuracy of recovery techniques? . . . 105
5.5.2 RQ2: What is the impact of different input factors, such as the granularity level, the use of transitive dependencies, the use of different symbol dependencies and dynamic-bindings graph construction algorithms, on existing architecture recovery techniques? . . . 109
5.5.3 RQ3: Can existing architecture recovery techniques scale to large projects comprising 10MSLOC or more? . . . 114
5.5.4 Comparison with Baseline Algorithms . . . 115
5.5.5 Summary of Results . . . 117
5.5.6 Comparison with the Prior Work . . . 117

5.6 Threats to Validity . . . 117
5.6.1 Metrics Limitations . . . 117
5.6.2 Selecting Metrics and Recovery Techniques . . . 119
5.6.3 Non-uniqueness of Ground-Truth Architectures . . . 119
5.6.4 Number of Projects Studied . . . 120
5.7 Summary . . . 120

6 Future Work **122**
6.1 Automatic Program Repair: Better Abstraction for Scalable NMT-based program repair . . . 122
6.2 Detecting Bugs in PDF Documents and Readers . . . 124
6.3 Software Architecture Recovery . . . 125
6.4 Explainable Defect Prediction: . . . 126

7 Conclusion **127**

References **128**

List of Figures

3.1 Two bugs in `Defects4J` fixed by CoCoNuT that other tools did not fix. . . 17
3.2 CoCoNuT's overview . . . 19
3.3 The new NMT architecture used in CoCoNuT. . . . 23
3.4 CoCoNuT's Java Patch for Lang 29 in `Defects4J` that have not been fixed by other techniques. . . . 33
3.5 Example of patches fixed only by CoCoNuT and related changes in the training set. . . . 34
3.6 BREADTH_FIRST_SEARCH bugs fixed by CoCoNuT in both Python and Java `QuixBugs` benchmarks. . . . 34
3.7 Number of bugs fixed as the number of models considered in ensemble learning increases. . . . 37
3.8 CoCoNuT's Java patch for Lang 26 in `Defects4J`. . . . 37
3.9 Attention map for the correct patch of *Lang 26* from the `Defects4J` benchmark generated by CoCoNuT. . . . 38

4.1 An example of a bug in `Firefox`. `Chromium` rendering on the left, `Firefox` rendering on the right. . . . 42
4.2 Overview of our approach, which consists of three main phases: the filtering phase (Section 4.2.1), the inconsistency detection phase (Section 4.2.2), and the inconsistency localization phase (Section 4.2.3). . . . 45
4.3 An example of a missing image bug in `MuPDF`. `MuPDF` on the right, other readers on the left. . . . 60
4.4 Example of colour discrepancy between `Acrobat Reader` (left) and other readers (right) . . . 61

4.5 An example of “other" types of inconsistencies. `Chromium` rendering on the left, distorted `Evince` rendering on the right. 62
4.6 `Chromium` (left) can render the incorrectly embedded characters, while other readers (right) cannot. 62
4.7 ROC curves for different image similarity algorithms. 65
4.8 Inconsistency between Adobe Reader (left) and Okular (right) before (a) and after reduction (b). 70

5.1 Overview of our approach 80
5.2 Example Project Layout 84
5.3 Example Project Submodules 84

6.1 Bug, abstraction, and mapping for the HANOI bug in `QuixBugs` that CoCoNuT cannot fix without using abstracted template. 123

List of Tables

3.1 Training set information. 27

3.2 Comparison with state-of-the-art G&V approaches. The number of bugs that only CoCoNuT fixes is in parentheses **(307)**. The results are displayed as x/y, with x the number of bugs correctly fixed. and y the number of bugs with plausible patches. * indicates tools whose manual fix patterns are used by TBAR. Numbers are extracted from either the original papers or from previous work [142] which reran some approaches with perfect localization. † indicates the number of patches that are identical to developer patches—the minimal number of correct patches. - indicates tools that have not been evaluated on a specific benchmark. The highest number of correct patches for each benchmark is in bold. 30

4.1 Basic statistics, extracted using `pdfinfo`, for the `GovDoc1` data set; `pdfinfo` failed for 15 documents. 51

4.2 Numbers of documents in our data set for various versions of the PDF standard. 53

4.3 Portfolio of PDF readers used in each phase. 55

4.4 Number of consistent and inconsistent files. For the latter, we report how many readers behave similarly (“agree”). 58

4.5 Issues detected in the random sample, sorted by type and reader. Acrobat denotes `Acrobat Reader`. 59

4.6 Number of crashes detected. The number of unique errors is shown in parentheses. 66

4.7 Inconsistencies for cluster candidates. The third column shows the number of inconsistent cluster candidates. The number of unique bugs is shown in parentheses. 66

4.8 Comparison between the distribution of files across PDF versions in the entire random sample and in the inconsistent files from the random sample. 68

4.9 Comparison between bugs on Linux and Windows version of PDF readers. # Linux column shows the number of bugs reported on Linux. # Windows represents the number and percentage of the reported bugs that also occur in the Windows version of the reader 72

4.10 Number of reported inconsistent files on Linux that also reveal bugs in Windows readers. 72

5.1 Evaluated projects and architectures. †Cluster denotes the number of clusters in the ground-truth architectures. N/A means the value is not available. . 87

5.2 MoJoFM results for Bash. 95

5.3 MoJoFM results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out. 95

5.4 MoJoFM results for ITK. 96

5.5 MoJoFM results for ArchStudio. 96

5.6 MoJoFM results for Hadoop. 97

5.7 a2a results for Bash. 97

5.8 a2a results for ITK. 98

5.9 a2a results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out. 98

5.10 a2a results for ArchStudio. 99

5.11 a2a results for Hadoop. 99

5.12 Normalized TurboMQ results for Bash. 100

5.13 Normalized TurboMQ results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out. 100

5.14 Normalized TurboMQ results for ITK. 101

5.15 Normalized TurboMQ results for ArchStudio. 101

5.16 Normalized TurboMQ results for Hadoop. 102

5.17 $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Bash. 102

5.18 $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ITK. 103

5.19 $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out. 103

5.20 $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ArchStudio. 104

5.21 $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Hadoop. 104

5.22 Wilcoxon Signed Rank when comparing different types of dependencies. . . 108

5.23 Cliff's δ Effect Size Tests. A negative value indicates that the effect size favor of the first dependency type listed. 108

5.24 Wilcoxon Signed Rank for each algorithm when compared to K-means . . . 116

5.25 Cliff's δ Effect Size Tests. A negative value indicates that the effect size is in favor of K-means. 116

Chapter 1

Introduction

Dependability is an important quality of modern software but is challenging to achieve. Many software dependability techniques have been proposed to help developers improve software reliability and dependability such as defect prediction [83, 96, 249], bug detection [6, 17, 146], program repair [51, 127, 150, 209, 261, 263], test case prioritization [152, 250], or software architecture recovery [13, 42, 67, 111, 164, 240].

Machine learning and deep learning have been used with great success to automate many tasks in many different domains such as autonomous driving cars [31] or diabetic blood glucose prediction [177].

In this thesis we consider how machine learning and deep-learning can be used to enhanced software dependability through three examples of different software dependability tasks: automatic program repair, bug detection in electronic document reader, and software architecture recovery.

In the first work, we show how one can use deep learning to learn from buggy and correct code snippets to automatically generate patches and repair software bugs (Chapter 3). In the second work, we use ML to cluster warning and error messages in order to select bug-revealing documents and detect more bugs in electronic document readers (Chapter 4). Finally, in the third work, we evaluate the effectiveness of different clustering approaches on software architecture recovery using different types of dependencies (Chapter 5).

The new techniques leveraging machine learning algorithms helped find 33 new bugs in PDF readers and automatically repair 493 bugs in popular benchmarks, 307 of which that had not been repaired automatically before.

1.1 Project 1 – Combining Context-Aware Neural Translation Models using Ensemble for Automated Program Repair:

Automated generate-and-validate (G&V) program repair techniques (APR) typically rely on hard-coded rules, thus only fixing bugs following specific fix patterns. These rules require a significant amount of manual effort to discover and it is hard to adapt these rules to different programming languages.

To address these challenges, we propose a new G&V technique—CoCoNuT, which uses *ensemble* learning on the combination of convolutional neural networks (CNNs) and a new context-aware neural machine translation (NMT) architecture to automatically fix bugs in multiple programming languages. To better represent the context of a bug, we introduce a new context-aware NMT architecture that represents the buggy source code and its surrounding context separately. CoCoNuT uses CNNs instead of recurrent neural networks (RNNs), since CNN layers can be stacked to extract hierarchical features and better model source code at different granularity levels (e.g., statements and functions). In addition, CoCoNuT takes advantage of the randomness in hyperparameter tuning to build multiple models that fix different bugs and combines these models using ensemble learning to fix more bugs.

Our evaluation on six popular benchmarks for four programming languages (Java, C, Python, and JavaScript) shows that CoCoNuT correctly fixes (i.e., the first generated patch is semantically equivalent to the developer's patch) 493 bugs, including 307 bugs that are fixed by none of the 27 techniques with which we compare.

Summary: This project supports the thesis by proposing a **new neural machine translation architecture for automatic program repair**. This technique fixes 493 bugs in 6 different benchmarks for four programming languages.

1.2 Project 2 – On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files:

Electronic documents are widely used to store and share information such as bank statements, contracts, articles, maps and tax information. Many different applications exist for displaying

a given electronic document, and users rightfully assume that documents will be rendered similarly independently of the application used. However, this is not always the case, and these inconsistencies, regardless of their causes—bugs in the application or the file itself—can become critical sources of miscommunication.

We present a study on the correctness of PDF documents and readers. We start by manually investigating a large number of real-world PDF documents to understand the frequency and characteristics of cross-reader inconsistencies, and find that such inconsistencies are common—13.5% PDF files are inconsistently rendered by at least one popular reader. We then propose an approach to detect and localize the source of such inconsistencies automatically.

We evaluate our automatic approach on a large corpus of over 230K documents using 11 popular readers and our experiments have detected 30 unique bugs in these readers and files. We also reported 33 bugs, some of which have already been confirmed or fixed by developers.

Summary: This project supports the thesis by using **k-means clustering** to select input PDF files to test and **ensure input diversity**. K-means clustering increases the percentage of bug revealing PDF files tested from 13.5% to 38.8%. This project helps to detect 30 unique bugs in popular readers and files.

1.3 Project 3 – Measuring the Impact of Dependencies on Software Architecture Recovery Techniques:

Many techniques have been proposed to automatically recover software architectures from software implementations. A thorough comparison among the recovery techniques is needed to understand their effectiveness and applicability.

This study improves on previous studies in two ways. First, we study the impact of leveraging accurate symbol dependencies on the accuracy of architecture recovery techniques. In addition, we evaluate other factors of the input dependencies such as the level of granularity and the dynamic-bindings graph construction. Second, we recovered an architecture of a large system, Chromium, that was not available previously. Obtaining a ground-truth architecture of Chromium involved two years of collaboration with its developers. As part of this work, we developed a new submodule-based technique to recover preliminary versions of ground-truth architectures.

The results of our evaluation of nine architecture recovery techniques and their variants suggest that (1) using accurate symbol dependencies has a major influence on recovery

quality, and (2) more accurate recovery techniques are needed. Our results show that some of the studied architecture recovery techniques scale to very large systems, whereas others do not.

Summary: This project supports the thesis by evaluating **machine learning-based clustering techniques for automatic software architecture recovery**. This project shows that the quality of the input dependencies fed to these techniques is paramount for high-quality recovery.

1.4 Publications

Earlier versions of the work presented in this thesis have been published in the following papers (listed in reversed chronological order). In these studies, my role included proposing novel ideas, collecting data, conducting experiments, analysing the results, and writing.

- **Thibaud Lutellier**, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan, *CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair*, Proceeding of The ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020 [157].
- Tomasz Kuchta*, **Thibaud Lutellier***, Edmund Wong, Lin Tan, and Cristian Cadar, *On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files*, (* The first two authors contributed equally to this paper), Empirical Software Engineering, 2018 [118].
- **Thibaud Lutellier**, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger, *Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques*, IEEE Transactions on Software Engineering, 2017 [156].
- **Thibaud Lutellier**, Devin Chollack, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic and Robert Kroeger, *Comparing Software Architecture Recovery Techniques Using Accurate Dependencies,* Proceeding of ACM/IEEE International Conference on Software Engineering, SEIP track, 2015 [154].

The following papers were published in parallel to the above mentioned publications. These studies are not directly related to this thesis, but explore other usages of machine learning or applications of software dependability approaches for machine learning.

- Hung Viet Pham, Shangshu Qian, Jiannan Wang, **Thibaud Lutellier**, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan, *Problems and Opportunities in Training Deep-Learning Software Systems: An Analysis of Variance.* Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. (ACM SIGSOFT Distinguished Paper Award) [198].

- Hung Viet Pham, **Thibaud Lutellier**, Weizhen Qi, and Lin Tan, CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. Proceeding of ACM/IEEE International Conference on Software Engineering, 2019 [197].

- Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine, **Thibaud Lutellier**, *Effective detection of vulnerable and malicious browser extensions.* Computers & Security, 2014 [219].

- Hossain Shahriar, Komminist Weldemariam, **Thibaud Lutellier**, Mohammad Zulkernine, *A Model-Based Detection of Vulnerable and Malicious Browser Extensions*. Proceedings of the 7th International Conference on Software Security and Reliability, 2013 [218].

Availability: Artifacts were made available on https://zenodo.org/record/4147418#.X5mI1HVKhhE.

Chapter 2

Related Work

2.1 Neural Machine Translation Models for APR

2.1.1 Deep Learning for Automatic Program Repair:

SequenceR [33], DLFix [134], and Tufano et al. [237] are the closest work related to CoCoNuT. The main differences with CoCoNuT are that these approaches use RNN (a single LSTM-based NMT model for SequenceR and Tufano et al., and a TreeRNN architecture for DLFix) and represent both the buggy line and its context as one input. These approaches have trouble extracting long term relations between tokens and do not capture the diversity of bug fixes. We show in Section 3.5.3 that CoCoNuT outperforms both DLFix and SequenceR. Tufano et al. [237] generates templates instead of complete patches and thus cannot be directly compared. Deep learning has also been used to detect and repair small syntax [213] and compilation [79, 80, 176] issues (e.g., missing parenthesis), or build errors [233]. These models show promising results for fixing compilation issues but only learn the syntax of the programming language. Other work fixes bugs in Karel, a beginner programming language [78]. Vasic et al. [241] uses deep learning to automatically localize and repair variable misuses.

Previous work also uses deep learning to represent source code and choose which graph modification to execute [48]. This work is different from our approach since it does not use neural machine translation. Ding et al. [50] studied the advantages and disadvantages of neural machine translation compared to deep learning classification models for APR.

2.1.2 G&V Program Repair:

Many APR techniques have been proposed [16, 32, 87, 94, 125, 127, 143, 147, 150, 167, 191, 209, 252, 259, 260, 261]. We use a different approach compared to these techniques, and as shown in Section 3.5, our approach fixes bugs that existing techniques have not fixed. In addition, these techniques require significant domain knowledge and manually crafted rules that are language-dependent, while thanks to our context-aware ensemble NMT approach, CoCoNuT automatically learns such patterns and is generalizable to several programming languages with minimal effort.

2.1.3 Grammatical Error Correction (GEC):

Recent work uses machine translation to fix grammar errors [34, 35, 69, 70, 100, 103, 145, 189, 211, 215, 267]. Among them, [35] applied an attention-based convolutional encoder-decoder model to correct sentence-level grammatical errors. CoCoNuT is a new application of NMT models on source code and programming languages, addressing unique challenges. Studying whether our new context-aware NMT architecture improves GEC remains future work.

2.1.4 Deep Learning in Software Engineering:

The software engineering community had applied deep learning to perform various other tasks such as defects prediction [130, 244, 249], source code representation [9, 11, 196, 245], source code summarization [10, 77] source code modeling [8, 29, 85, 255], code clone detection [131, 254], program synthesis [7, 136, 182, 186], and fault localization [273, 274].

2.2 Finding, and Localizing Inconsistency Bugs in PDF Readers and Files

2.2.1 Cross PDF Reader Inconsistency:

To address cross-reader inconsistencies, standards such as PDF/A and PDF/X have been proposed to ensure that PDF files will always be rendered consistently [90, 91]. However, those specifications are highly restrictive and typically only used for archiving data. For example, when looking at the sample of PDF files we extracted from the `Govdocs1` data set,

we found that none of these files conformed to the PDF/A or PDF/X standards. Another possible solution to reduce display differences is flattening the PDF file. However, flattening a PDF file may add more inconsistencies when merging the different layers of the original file. In addition, flattening may break the internal structure of the file, making some operations (e.g., selecting the text) impossible.

Therefore, a new solution is necessary to help users know whether the PDF they share will be viewed correctly across the readers.

2.2.2 Cross Browser Inconsistency:

The cross-reader inconsistency issue is analogous to the more studied cross-browser inconsistency (CBI) problem [36, 37, 38, 52, 175, 192, 206, 207]. Different browsers can render the same webpage differently, and some of those inconsistencies are critical when a browser does not support a particular HTML element.

Eaton and Memon first introduced the cross-browser inconsistency (CBI) issue in [52]. They propose an approach based on an inductive model to detect bugs in web application that can lead to CBIs. This first approach to detect CBIs only focus on incorrect web applications. If a bug in a specific browser creates an inconsistency for a correct web application, then this approach would not detect it. In addition, this approach is based on HTML tags and is not transferable to the cross-reader inconsistency problem.

The CBI problem has been addressed incrementally by Choudhary et al. in several papers [36, 37, 38, 206]. First, Webdiff, described in [38] and [36], combines a structural analysis of the DOM structure of a web page and a histogram comparison of screenshots of the web page opened in two different browsers to detect CBIs. CrossCheck [37] is built on the top of WebDiff, with the addition of a web crawler allowing to detect inconsistencies on entire web applications instead of single web pages. Finally, X-pert [206] improves CrossCheck by only considering elements that are leaf nodes of the DOM structure. It builds a model for each browser by crawling a specific web application. The models include transitions between pages, as well as the DOM structure and a screenshot of each page. The models are then checked for equivalence using the chi-square distance between the histogram of each element of the screenshot.

Concurrently with WebDiff [36,38], Mesbah and Prasad proposed a very similar approach to detect cross-browser inconsistencies [175]. The main difference with WebDiff is that they consider the trace-level behavior of the web application.

Different causes for CBI were identified in [192]. The most common reasons for CBI are HTML tags, CSS, font rendering, DOM, scripts, add-ons and third-party entities. Out

of all these reasons, only the font rendering issue is also applicable to the cross-reader inconsistency issue.

While the importance of studying cross-browser inconsistencies has been widely recognized, cross-reader inconsistencies have been under-studied. Furthermore, some of the techniques used to detect cross-browser inconsistency are not directly applicable to cross-reader inconsistencies (for example, they rely on matching the DOM, the structured representation of HTML documents, but the structure of a PDF does not change when opened by different readers). On the other hand, our approach is applicable to a wide variety of electronic documents, including HTML, because it treats the documents and readers as black boxes.

The work from the cross-browser testing area that is most closely related to ours is `Browserbite` [207]. As in our approach, the technique operates in a black-box manner; it combines image processing for inconsistency detection with machine learning for improving accuracy. However, in addition to the different domain, `Browserbite` is evaluated on only 140 websites, which does not raise the same scalability challenges that we encountered for our corpus of 230K documents, which requires solutions such as our clustering approach. We use a white-box binary search to find inconsistent locations, while `Browserbite` splits image into regions and performs a linear comparison of all regions; we also use CW-SSIM for image comparison, while `Browserbite` uses histograms.

2.2.3 Vulnerabilities in PDF files:

Many techniques have been developed to detect malicious or vulnerable PDF files [41, 43, 53, 93, 104, 105, 120, 121, 132, 161, 214, 223, 239]. Indeed, PDF files can contain embedded JavaScript elements which can be potentially vulnerable or malicious. Common methods to detect such files consist of analysing the metadata and the structure of the PDF file to detect malicious JavaScript components.

MDScan [239] is a stand-alone tool that combines static analysis of the document and dynamic analysis to detect malicious documents. First, a static analysis of the PDF file is used to detect and extract any embedded JavaScript source code. Then this code is executed and MDScan attempts to detect malicious shellcode execution.

PJScan [121] is the first approach using static analysis to detect vulnerable PDF files. It focuses on extracting features from Javascript code embedded in PDF files. Smutz et al. [223] present another approach based on machine learning that aims to detect vulnerabilities. The features used to detect malicious PDF files are extracted from the metadata and the structure of the PDF (e.g., number of pages, objects, producer, javascript elements, etc.).

This approach was compared to PJScan and provided much better results. An improvement of existing static detection [121, 223] was proposed in [120]. The main change was to consider features in object streams. As object streams are generally encoded, they are often used to hide malicious code. While machine learning could also be used to detect potential inconsistent elements, this approach is very different from our work. In this work, we chose a dynamic approach: both the error clustering and the screenshot comparison require executing the PDF reader.

Maiorca et al. [161] present a new approach to generate malicious PDF files that cannot be detected by techniques based on machine learning and propose a new tool, Lux0R [43] to detect this kind of attack.

While dealing with PDF files, those studies consider a completely different problem from ours. We focus on benign documents and do not consider malware. Our technique, through detecting display and behavior inconsistencies across different readers, has the potential to identify malicious PDF targeting a specific PDF reader. However, we did not investigate this possibility and assumed that all the files mined from the US government websites are benign.

2.2.4 Differential Testing:

Differential testing [168] consists of testing whether different software produce the same results. Much work uses differential testing to find bugs in compilers by comparing the output of multiple compilers [220, 265, 268], different compiler optimization levels [124, 265], or differences in deep learning APIs [197]. Inconsistency detection has been used in other domains such as cross-platform [57, 99]. Our work is a new application of differential testing and inconsistency detection for findings bugs in electronic documents and readers.

2.2.5 Document Recovery:

Prior work on recovering electronic documents [47, 117, 149] can leverage our inconsistency detection and localization techniques. Previous work in this space uses image similarity to assess the quality of repaired input [149].

2.3 Software Architecture Recovery Techniques

2.3.1 Comparison of Software Architecture Recovery Techniques:

This paper builds on work that was previously reported in [155]. Novelty with respect to this previous work includes a study of the impact of dynamic-bindings resolution algorithms, function calls, and global variable usage on the accuracy of recovery algorithms. We also expand the previous work by studying whether using a higher level of granularity and transitive dependencies improves the accuracy of recovery techniques.

Many architecture recovery techniques have been proposed [13, 42, 67, 111, 164, 187, 240]. The most recent study [64] collected the ground-truth architectures of eight systems and used them to compare the accuracy of nine variants of six architecture recovery techniques. Two of those recovery techniques—Architecture Recovery using Concerns (ARC) [67] and Algorithm for Comprehension-Driven Clustering (ACDC) [240]—routinely outperformed the others. However, even the accuracy of these techniques showed significant room for improvement.

Architecture recovery techniques have been evaluated against one another in many other studies [13, 64, 111, 128, 137, 165, 225, 229, 256].

The results of the different studies are not always consistent. scaLable InforMation BOttleneck (LIMBO) [12], a recovery technique leveraging an information loss measure, and ACDC performed similarly in one study [13]; however, in a different study, Weighted Combined Algorithm (WCA) [166], a recovery technique based on hierarchical clustering, outperformed Complete Linkage (CL) [166]. In yet another study, CL is shown to be generally better than ACDC [256]. In yet another study [64], ARC and ACDC surpass LIMBO and WCA. Wu et al. [256] compared several recovery techniques utilizing three criteria: stability, authoritativeness, and non-extremity. For this study, no recovery technique was consistently superior to others on multiple measures. A possible explanation for the inconsistent results of these studies is their use of different assessment measures.

The types of dependencies which serve as input to recovery techniques vary among studies: some recovery techniques leverage control and data dependencies [59, 60, 235]; other techniques use static and dynamic dependency graphs [13].

Previous work [203] examined the effect of different polymorphic call-graph construction algorithms on automatic clustering. Another work [84] studied the impact of source-code versus object-code-based dependencies on software architecture recovery. They found that dependencies obtained directly from source code are more useful than dependencies obtained

from object code. While also studying the impact of dependencies on automatic architecture recovery, we focus on the accuracy and the type of the dependencies (e.g., include and symbol dependencies) independently from the way dependencies were extracted, i.e., from object code or source code.

None of the papers mentioned above assess the influence of symbol dependencies on recovery techniques when compared to include dependencies. This paper is the first to study (1) the impact of symbol dependencies on the accuracy of recovery techniques and (2) the scalability of recovery techniques to a large project with nearly 10MSLOC.

2.3.2 Recovery of Ground-Truth Architectures:

Ground-truth architectures enable the understanding of implemented architectures and the improvement of automated recovery techniques. Several prior studies invested significant time and effort to recover ground-truth architectures for several systems.

Garcia et al. [65] described a method to recover the ground-truth architectures of four open-source systems. The method involves extensive manual work, and the mean cost of recovering the ground-truth architecture of seven systems ranged from 70KSLOC to 280KSLOC was 107 hours. CacOphoNy [55] is another approach that uses metamodels to aid in manual recovery of software architectures and had been used to reverse engineer a large software system [56].

In his work [119], Laine manually recovered the architecture of the X-Window System to illustrate the importance of software architecture for object-oriented development.

Grosskurth et al. [76] studied the architecture and evolution of web browsers and provide guidance for obtaining a reference architecture for web browsers. Their work does not address the challenges of recovering an accurate ground-truth architecture in general. In addition, it is not clear if their approach is accurate for modern web browsers such as Chromium, which use new design principles such as a modern threading model for tabbed browsing.

Bowman et al. [23] and Xiao et al. [257] recovered the ground-truth architectures of the Linux kernel 2.0 and Mozilla 1.3 respectively. The Linux kernel and Mozilla are large systems, but the evaluated versions are more than a decade old. The version of the Linux kernel recovered was from 1996 and at that time, it contained only 750KSLOC. Mozilla 1.3 is from 2003 with 4MSLOC.

Several tools have been created to help developers analyze, visualize, and reconstruct software architectures. [190] Those tools can be used in different stages of manual architecture

recovery.

Rigi [226] is a tool that can be used to analyse and visualize software dependencies. While it is possible to generate an architectural view of a project with the help of Rigi [107], this requires the intervention of a developer with deep knowledge of the project to manually group similar elements (classes, files, etc.) together. Indeed, for large projects, initial views proposed by Rigi are unreadable due to the large number of nodes and dependencies [106] and manual effort is necessary to recover the architecture of the system. The Portable Bookshelf [58], SHriMP [184], AOVIS [112], LSEdit [228] are other software visualization and analysis tools that can help manual architecture recovery.

Several other tools such as Understand [1], Lattix [243] and Structure101 [30] have been used to ensure the quality of a given architecture and monitor its evolution. However, none of these tools intend to automatically recover an architecture.

Chapter 3

CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair

3.1 Motivation

To improve software reliability and increase engineering productivity, researchers have developed many approaches to fix software bugs automatically. One of the main approaches for automatic program repair is the G&V method [51,127,150,209,261,263]. First, candidate patches are generated using a set of transformations or mutations (e.g., deleting a line or adding a clause). Second, these candidates are ranked and validated by compiling and running a given test suite. The G&V tool returns the highest-ranked fix that compiles and passes fault-revealing test cases.

While G&V techniques successfully fixed bugs in different datasets, a recent study [151] showed that very few correct patches are in the search spaces of state-of-the-art techniques, which puts an upper limit on the number of correct patches that a G&V technique can generate. Also, existing techniques require extensive customization to work across programming languages since most fix patterns need to be manually re-implemented to work for a different language.

Therefore, there is a need for a new APR technique that can fix more bugs (i.e., with a better search space) and is easily transferable to different programming languages.

Neural machine translation is a popular deep-learning (DL) approach that uses neural

network architectures to generate likely sequences of tokens given an input sequence. NMT uses an *encoder* block (i.e., several layers of neurons) to create an intermediate representation of the input learned from training data and a *decoder* block to decode this representation into the target sequence. NMT has mainly been applied to natural language translation tasks (e.g., translating French to English).

APR can be seen as a translation from buggy to correct source code. Therefore, there is a unique opportunity to apply NMT techniques to learn from the readily available bug fixes in open-source repositories and generate fixes for unseen bugs.

Such NMT-based APR techniques have two key advantages. First, NMT models automatically learn complex relations between input and output sequences that are difficult to capture manually. Similarly, NMT models could also capture complex relations between buggy and clean code that are difficult for manually designed fix patterns to capture. Second, while G&V methods often use hard-coded fix patterns that are programming-language-dependent and require domain knowledge, NMT techniques can be retrained for different programming languages automatically without re-implementing the fix patterns, thus requiring little manual effort.

Despite the great potential, there are two main challenges of applying NMT to APR:

(1) Representing context: How to fix a bug often depends on the context, e.g., statements before and after the buggy lines. However, to represent the context effectively is a challenge for NMT models in both natural language tasks and bug fixing tasks; thus, the immediate context of the sentence to be translated is generally ignored. Two techniques that use NMT to repair bugs [33,237] concatenate context and buggy code as one input instance. This design choice is problematic. First, concatenating context and buggy code makes the input sequences very long, and existing NMT architectures are known to struggle with long input. As a result, such approaches only fix short methods. For example, Tufano et al. [237] have to focus on short methods that contain fewer than 50 tokens. Second, concatenating buggy and context lines makes it more difficult for NMT to extract meaningful relations between tokens. For example, if the input consists of 1 buggy line and 9 lines of context, the 9 context lines will add noise to the buggy line and prevent the network from learning useful relations between the buggy line and the fix, which makes the models inefficient and less effective on fixing the buggy code.

We propose a new **context-aware NMT architecture** that has two separate encoders: one for the buggy lines, the other one for the context. Using separate encoders presents three advantages. First, the buggy line encoder will only have shorter input sequences. Thus, it will be able to extract strong relations between tokens in the buggy lines and the correct fix without wasting its resources on relations between tokens in the context. Second,

a separate context encoder helps the model learn useful relations from the context (such as potential donor code, variables in scope, etc.) without adding noise to the relations learned from the buggy lines. Third, since the two encoders are independent, the context and buggy line encoders can have different complexity (e.g., different number of layers), which could improve the performance of the model. For example, since the context is typically much longer than the buggy lines, the context encoder may need larger convolution kernels to capture long-distance relations, while the buggy line encoder may benefit from a higher number of layers (i.e., a deeper network) to capture more complex relations between buggy and clean lines. *This context-aware NMT architecture is novel and can also be applied to improve other tasks such as fixing grammar mistakes and natural language translation.*

(2) Capturing the diversity of bug fixes: Due to the diversity of bugs and fixes (i.e., many different types of bugs and fixes), a single NMT model using the "best" hyperparameters (e.g., number of layers) would struggle to generalize.

Thus, we leverage **ensemble learning** to combine models of different levels of complexity that capture different relations between buggy and clean code. This allows our technique to learn diverse repair strategies to fix different types of bugs.

In this work, we propose a new G&V technique called **CoCoNuT** that consists of an ensemble of fully convolutional (FConv) models and new context-aware NMT models of different levels of complexity. Each context-aware model captures different information about the repair operations and their context using two separate encoders (one for buggy lines and one for the context). Combining such models allows CoCoNuT to learn diverse repair strategies that are used to fix different types of bugs while overcoming the limitations of existing NMT approaches.

Evaluated against 27 APR techniques on six bug benchmarks in four programming languages, CoCoNuT fixes 493 bugs, 307 of which have not been fixed by any existing APR tools. Figure 3.1 shows two of such patches for bugs in `Defects4J` [101], demonstrating CoCoNuT's capability of learning *new* and *complex* relations between buggy and clean code from the 3,241,966 instances in the Java training set. CoCoNuT generates the patch in Figure 3.1a using a **new pattern** (i.e., updating the Error type to be caught) that previous work [32, 87, 94, 115, 125, 127, 138, 139, 140, 141, 143, 167, 200, 201, 209, 210, 252, 259, 260] did not discover. In the Figure, "-" denotes a line to be deleted that is taken as input by CoCoNuT, while "+" denotes the line generated by CoCoNuT and is identical to the developer's patch. These previous techniques represent **a decade of APR research** that uses manually-designed Java fix patterns. This demonstrates that CoCoNuT complements existing APR approaches by automatically learning new fix patterns that existing techniques did not have, and automatically fixing bugs that existing techniques did not fix. To fix the bug in

```
- catch (org.mockito.exceptions.verification.junit.
     ArgumentsAreDifferent e) {
+ catch (AssertionError e) {
```

(a) Patch for Mockito 5 in `Defects4J` for Java.

```
  if (actualTypeArgument instanceof WildcardType) {
      contextualActualTypeParameters.put(typeParameter,
         boundsOf((WildcardType) actualTypeArgument));
- } else {
+ } else if (typeParameter != actualTypeArgument) {
      contextualActualTypeParameters.put(typeParameter,
         actualTypeArgument);
  }
```

(b) Patch for Mockito 8 only fixed by a context-aware model.

Figure 3.1: Two bugs in `Defects4J` fixed by CoCoNuT that other tools did not fix.

Figure 3.1b, CoCoNuT learns from the context lines the correct variables (`typeParameter` and `actualTypeArgument`) to be inserted in the conditional statement.

This work makes the following contributions:

- A new *context-aware NMT* architecture that represents context and buggy input separately. This architecture is independent of our fully automated tool and could be applied to solve general problems in other domains where long-term context is necessary (e.g., grammatical error correction).
- The first application of *CNN* (i.e., FConv [71] architecture) for APR. We show that the FConv architecture outperforms LSTM and Transformer architectures when trained on the same dataset.
- An *ensemble* approach that combines context-aware and FConv NMT models to better capture the diversity of bug fixes.
- CoCoNuT is the first APR technique that is *easily portable* to different programming languages. With little manual effort, we applied CoCoNuT to four programming languages (Java, C, Python, and JavaScript), thanks to the use of NMT and new tokenization.
- A thorough evaluation of CoCoNuT on six benchmarks in four programming languages. CoCoNuT fixes 493 bugs, 307 of which have not been fixed by existing APR tools.

- A use of attention maps to explain why certain fixes are generated or not by CoCoNuT.

Artifacts are available[1]**.**

3.2 Background and terminology

Terminology: A DL *network* is a structure (i.e., a graph) that contains nodes or *layers* that are stacked to perform a specific task. Each type of layer represents a specific low-level transformation (e.g., convolution, pooling) of the input data with specific *parameters* (i.e., *weights*). We call DL *architecture* an abstraction of a set of DL networks that have the same types and order of layers but do not specify the number and dimension of layers. A set of *hyperparameters* specifies how one consolidates an architecture to a network (e.g., it defines the convolutional layer dimensions or the number of convolutions in the layer group). The hyperparameters also determine which optimizer is used in training along with the optimization parameters, such as the learning rate and momentum. We call a *model* (or trained model), a network that has been trained, which has fixed weights.

Attention: The attention mechanism [19] is a recent DL improvement. It helps a neural network to focus on the most important features. Traditionally, only the latest hidden states of the encoder are fed to the decoder. If the input sequence is too long, some information regarding the early tokens are lost, even when using LSTM nodes [19]. The attention mechanism overcomes this issue by storing these long-distance dependencies in a separate attention map and feeding them to the decoder at each time step.

Candidate, Plausible, and Correct Patches: We call patches generated by a tool *candidate patches*. Candidate patches that pass the fault triggering test cases are *plausible patches*. Plausible patches that are semantically equivalent to the developers' patches are *correct patches*.

3.3 Approach

Our technique contains three stages: training, inference, and validation. Figure 3.2 shows an overview of CoCoNuT. In the training phase, we extract tuples of buggy, context, and fixed lines from open-source projects. Then, we preprocess these lines to obtain sequences

[1] https://github.com/lin-tan/CoCoNut-Artifact

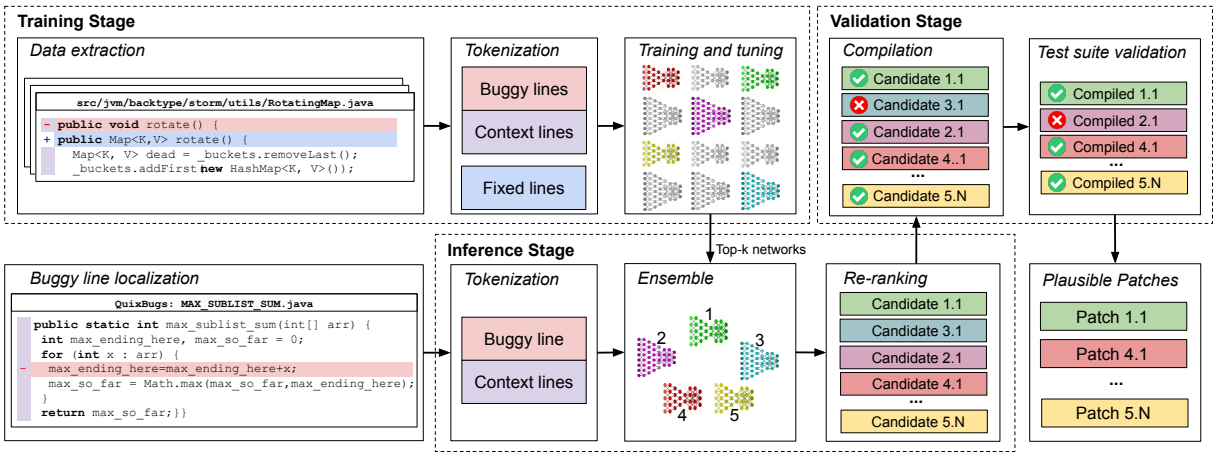


Figure 3.2: CoCoNuT's overview

of tokens, feed the sequences to an NMT network, and tune the network with different sets of hyperparameters. We further train the top-k models until convergence to obtain an ensemble of k models. Since each model has different hyperparameters, each model learns different information that helps fix different bugs.

In the inference phase, a user inputs a buggy line and its context into CoCoNuT. It then tokenizes the input and feeds it to the top-k best models, which each outputs a list of patches. These patches are then ranked and validated by compiling the patched project. CoCoNuT then runs the test suite on the compilable fixes to filter incorrect patches. The final output is a list of *candidate patches* that pass the validation stage.

Section 3.3.1 presents the challenges of using NMT to automatically fix bugs, while the rest of Section 3.3 describes the different components of CoCoNuT.

3.3.1 Challenges

In addition to the two main challenges discussed in Introduction, i.e., **(1) representing context** and **(2) capturing the diversity of fix patterns**, using NMT for APR has additional challenges:

(3) Choice of Layers of Neurons: While natural language text is read sequentially from left to right, source code is generally not executed in the same sequential way (e.g., conditional blocks might be skipped in the execution). Thus, relevant information can be

located farther away from the buggy location (e.g., a variable definition can be lines away from its buggy use). As a result, traditional recurrent neural networks (RNN) using LSTM layers [153] may not perform well for APR.

To address these challenges, we build our new context-aware NMT architecture using convolutional layers as the main component of the two encoders and the decoder, as they better capture such different dependencies than RNN layers [45, 71]. We stack convolutional layers with different kernel sizes to represent relations of different levels of granularity. Layers of larger kernel sizes model long-term relations (e.g., relations within a function), while layers of smaller kernel sizes model short-term relations (e.g., relations within a statement). To further track long-term dependencies in both input and context encoders, we use a multi-step attention mechanism.

(4) Large vocabulary size: Compared to traditional natural language processing (NLP) tasks such as translation, the vocabulary size of source code is larger and many tokens are infrequent because developers can practically create arbitrary tokens. In addition, letter case indicates important meanings in source code (e.g., `zone` is a variable and `ZONE` a constant), which increases the vocabulary size further. For example, previous work [33] had to handle a code vocabulary size larger than 560,000 tokens. Practitioners need to cut the vocabulary size significantly to make it scalable for NMT, which leaves a large number of infrequent out-of-vocabulary tokens. We address this challenge by using a new tokenization approach that reduces the vocabulary size significantly without increasing the number of out-of-vocabulary words. For Java, our tokenization reduces the vocabulary size from 1,136,767 to 139,423 tokens while keeping the percentage of tokens out-of-vocabulary in our test sets below 2% (Section 3.3.3).

3.3.2 Data Extraction

We train CoCoNuT on tuples of buggy, context, and fixed lines of code extracted from the commit history of open-source projects. To remove commits that are not related to bug fixes, we follow previous work [249] and only keep commits that have the keywords "fix," "bug," or "patch" in their commit messages. We also filter commits using six commit messages anti-patterns: "rename," "clean up," "refactor," "merge," "misspelling," and "compiler warning." We manually investigate a random sample of 100 commits filtered using this approach and 93 of them are bug-related commits. This is a reasonable amount of noise (7%) for ML training on large training data [109, 258]. We split commits into *hunks* (groups of consecutive differing lines) and consider each hunk as a unique *instance*.

We represent the context of the bug using the function surrounding the buggy lines

because it gives semantic information about the buggy functionality, is relatively small, contains most of the relevant variables, and can be extracted for millions of instances. The block "Data Extraction" in Figure 3.2 shows an example with one buggy line (highlighted in red), one correct line (highlighted in blue), and some lines of context (with purple on the left).

The proposed context-aware NMT architecture is independent of the choice of the context and would still work with different context definitions (e.g., full file or data-flow context). Our contribution is to represent the buggy line and the context separately as a second encoder, independently of the chosen context. Exploring other context definitions is interesting future work.

3.3.3 Input Representation and Tokenization

CoCoNuT has two separate inputs: a buggy line and its context. The block "Buggy line localization" in Figure 3.2 shows the inputs for the `MAX_SUBLIST_SUM` bug in QuixBugs [135]. The line highlighted in red is the buggy line and the lines with a purple block on the left (i.e., the entire function) represent the context. Since typical NMT approaches take a vector of tokens as input, our first challenge is to choose a correct abstraction to transform a buggy line and its context into sequences of tokens.

We use a tokenization method analogous to word-level tokenization (i.e., space-separated), a widely used tokenization in NLP. However, word-level tokenization presents challenges that are specific to programming languages. First, we separate operators from variables as they might not be space-separated. Second, the vocabulary size is extremely large and many words are infrequent or composed of multiple words without separation (e.g., getNumber and get_Number are two different words). To address this issue, we enhance the word-level tokenization by also considering underscores, camel letters, and numbers as separators. Because we need to correctly regenerate source code from the list of tokens generated by the NMT model, we also need to introduce a new token (`<CAMEL>`) to mark where the camel case split occurs. In addition, we abstract string and number literals except for the most frequent numbers (0 and 1) in our training set.

Thanks to these improvements, we reduce the size of the vocabulary significantly (e.g., from 1,136,767 to 139,423 tokens for Java), while limiting the number of out-of-vocabulary tokens (in our benchmarks, less than 2% of the tokens were out-of-vocabulary).

3.3.4 Context-Aware NMT Architecture

CoCoNuT's architecture presents two main novelties. The first one consists of using two separate encoders to represent the context and the buggy lines (Figure 3.3). The second is a new application of fully convolutional layers (FConv) [71] for APR instead of traditional RNN such as LSTM layers used in previous work [33]. We choose the FConv layers because FConv's CNN layers can be stacked to extract hierarchical features for larger contexts [45], which enables modeling source code at different granularity levels (e.g., variable, statement, block, and function). This is closer to how developers read code (e.g., looking at the entire function, a specific block, and then variables). RNN layers model code as a sequence, which is more similar to reading code from left to right.

We evaluate the impact of FConv and LSTM in Section 3.5.3.

Our architecture consists of several components: an input encoder, a context encoder, a merger, a decoder, and an attention module. For simplicity, Figure 3.3 only displays a network with one convolutional layer. In practice, depending on the hyperparameters, a complete network has 2 to 10 convolutional layers for each encoder and the decoder.

In training mode, the context-aware model has access to the buggy lines, its context, and the fixed lines. The model is trained to generate the best representation of the transformation from buggy lines with context to fixed lines. In practice, this is conducted by finding the best combination of weights that translates the input instances from the training set to fixed lines. Multiple passes on the training data are necessary to obtain the best set of weights.

In inference mode, since the model does not have access to the fixed line, the decoder processes tokens one by one, starting with a generic <`START`> token. The outputs of the decoder and the merger are then combined through the multi-step attention module. Finally, new tokens are generated based on the outputs of the attention, the encoders, and the decoder. The generated token is then fed back to the decoder until the <`END`> token is generated.

Following the example input in Figure 3.3, a user inputs the buggy statement "`int sum=0;`" and its context to CoCoNuT. After tokenization, the buggy statement (highlighted in red) is fed to the Input encoder while the context (highlighted in purple) is fed to the Context encoder. The outputs of both encoders are then concatenated in the Merger layer.

Since CoCoNuT did not generate any token yet, the token generation starts by feeding the token <`START`> to the decoder (iteration 0). The output of the decoder (d_{out}) is then combined with the merger output using a dot product to form the first column of the attention map. The colors of the attention map indicate how important each input token is

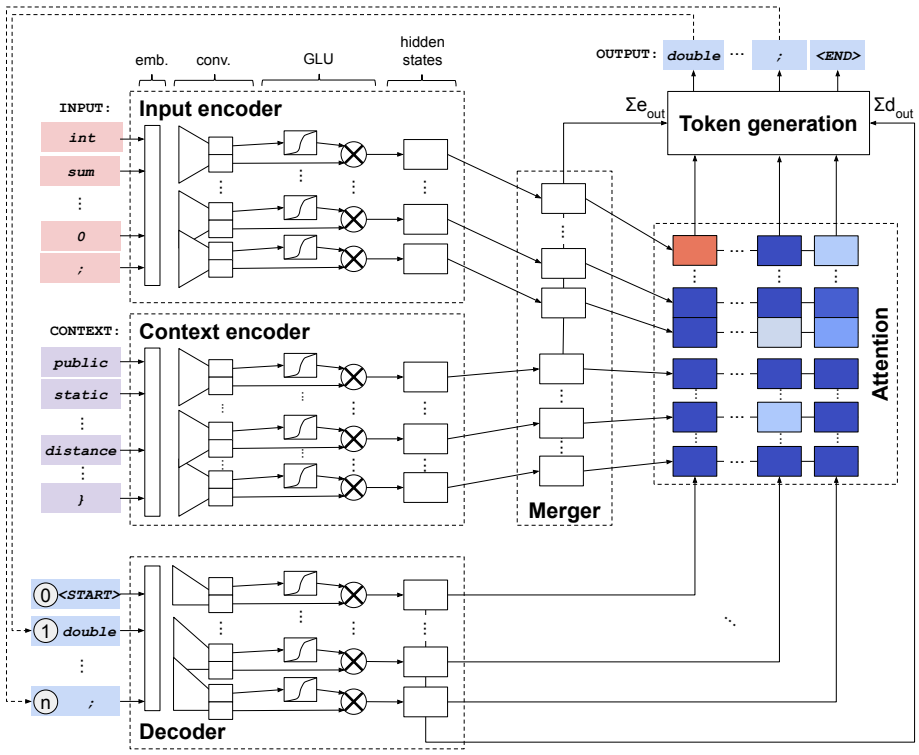


Figure 3.3: The new NMT architecture used in CoCoNuT.

for generating the output. For example, to generate the first token (`double`), the token `int` is the most important input token as it appears in red. The token generation combines the output of the attention, as well as the sum of the merger and decoder outputs ($\Sigma\ e_{out}$ and Σd_{out}) to generate the token `double`. This new token is added to the list of generated tokens and the list is given back as a new input to the decoder (iteration 1). The decoder uses this new input to compute the next d_{out} that is used to build the second column of the attention map and to generate the next token. The token generation continues until <`END`> is generated.

We describe the different modules of the network below.

Encoders and Decoder: The purpose of the encoders is to provide a fixed-length vectorized representation of the input sequence while the decoder translates such representation to the target sequence (i.e., the patched line). Both modules have a similar structure that consists of three main blocks: an embedding layer, several convolutional layers, and a layer of gated linear units (GLU).

The embedding layers represent input and target tokens as vectors, with tokens occurring in similar contexts having a similar vector representation. In a sense, these layers represent the model's knowledge of the programming language.

The output of the embedding layers is then fed to several convolutional layers. The size of the convolution kernel represents the number of surrounding tokens that are taken into consideration. Stacking such convolutional layers with different kernel sizes provides multiple levels of abstraction for our network to work with. For example, a layer with a small kernel size will only focus on a few surrounding tokens within a statement, while larger layers will focus on a block of code or even a full function. These layers are different in the encoder and the decoder. The encoders use information from both the previous and the next tokens in the input sequence (since the full input sequence is known at all times), while the decoder only uses information about the previously generated tokens (since the next tokens have not been generated yet). Figure 3.3 shows these differences (full triangles in the encoders and half triangles in the decoder).

After the convolutional layers, a layer of GLU (represented by the sigmoid and multiplication boxes in Figure 3.3) decides which information should be kept by the network. The Merger then concatenates the outputs of the input and context encoders.

Multi-Step Attention: Compared to traditional attention (see Section 3.2), multi-step attention uses an attention mechanism to connect the output of each convolutional layer in the encoders and the decoder. When multiple convolutional layers are used, it results in multiple attention maps. Multi-step attention is useful because it connects each level of abstraction (i.e., convolutional layer) to the output. This increases the amount of information an encoder passes to the decoder when generating the target tokens.

The attention map represents the impact of input tokens on generating a specific output token and can help explain why a specific output is generated. We analyze attention maps in RQ4.

Token Generation: The token generation combines the outputs of the attention layers, merger ($\Sigma\ e_{out}$) and decoder ($\Sigma\ d_{out}$) to generate the next token. Each token in the vocabulary is ranked by the token generation component based on their likelihood of being the next token in the output sequence. The token selected by the search algorithm is then appended to the list of generated tokens, and the list is sent back as the new input of the decoder. The token generation stops when the <END> token is generated.

Beam Search: We use a common search strategy called beam search to generate and rank a large number of candidate patches. For each iteration, the beam search algorithm checks the t most likely tokens (t corresponds to the beam width) and ranks them by the total

likelihood score of the next s prediction steps (s correspond to the search depth). In the end, the beam search algorithm outputs the top t most likely sequences ordered based on the likelihood of each sequence.

3.3.5 Ensemble Learning

Fixing bugs is a complex task because there are very diverse bugs with very different fixing patterns that vary in terms of complexity. Some fix patterns are very simple (e.g., changing the operator $<$ to $>$) while others require more complex modifications (e.g., adding a null checker or calling a different function). Training a model to fix all types of bugs is difficult. Instead, it is more effective to combine multiple specialized models into an ensemble model that will fix more bugs than one single model.

Therefore, we propose an ensemble approach that combines: (1) models with and without context, and (2) models with different hyperparameters (different networks) that perform the best on our validation set.

As described in Section 3.2, hyperparameters consolidate an architecture to a network. Different hyperparameters have a large impact on the complexity of a network, the speed of the training process, and the final performance of the trained model. For this tuning process, we apply random search because previous work showed that it is an inexpensive method that performs better than other common hyperparameter tuning strategies such as grid search and manual search [20]. For each hyperparameter, based on our hardware limitations, we define a range from which we can pick a random value. Since training a model until convergence is very expensive, we tune with only one epoch (i.e., one pass on the training data). We train n models with different random sets of parameters to obtain models with different behavior and keep the top k best models based on the performance of each model on a separate validation set. This tuning process allows us to discard underfit or overfit models while keeping the k best models that converge to different local optima and fix different real-world bugs.

Finally, we sort patches generated by different models based on their ranks. We use the likelihood of each sequence (i.e., bug fix) generated by each model to sort patches of equal ranks.

3.3.6 Patch Validation

Statement Reconstruction: A model outputs a list of tokens that forms a fix for the input buggy line. The statement reconstruction module generates a complete patch from

the list of tokens. For the abstracted tokens (i.e., strings and numbers), we extract donor code from the original file in which the bug occurred. Once the fix is generated, it is inserted at the buggy location, and we move to the validation step.

Compilation and Test Suite Validation: The model does not have access to the entire project; therefore, it does not know whether the generated patches are compilable or pass the test suite. The validation step filters out patches that do not compile or do not pass the triggering test cases. We use the same two criteria as previous work [263] for the validation process. First, the test cases that make the buggy version pass should still pass on the patched version. Second, at least one test case that failed on the buggy version should pass on the patched version.

3.3.7 Generalization to Other Languages

Since CoCoNuT learns patterns automatically instead of relying on handcrafted patterns, it can be generalized to other programming languages with minimum effort. The main required change is to obtain new input data. Fortunately, this is easy to do since our training set is extracted from open-source projects. Once the data for the new programming language has been extracted, the top k models can be retrained without re-implementation. CoCoNuT will learn fix patterns automatically for the new programming language.

3.4 Experimental Setup

Selecting Training Data: Since the earliest bugs in our Java benchmarks are from 2006 and the latest are from 2016, we divide the instances extracted from the training projects into two training sets. The first one contains all instances committed before 2006 and the second one contains instances committed before 2010. Instances committed after 2010 are discarded. The models trained using the first training set can be used to train models to fix all bugs in the test benchmarks, while the models trained using the second training set should only be used on the bugs fixed after 2010. This setup is to ensure the validity of experiments so that no future data is used [230] (Section 3.5).

For Java, we use GHTorrent [74] to collect instances from 59,237 projects that contain commits before 2010. We also extract data from the oldest 20,000 Gitlab projects (restriction caused by the limitation of Gitlab search API) because we expect older projects to contain more bugs fixed before the first bug in our benchmarks, and 10,111 Bitbucket repositories (ranked by popularity because of limitations of the Bitbucket search API).

Table 3.1: Training set information.

Language	# projects	# instances
Java 2006	45,180	3,241,966
Java 2010	59,237	14,796,149
Python 2010	13,899	480,777
C 2005	12,577	2,735,506
JavaScript 2010	10,163	3,217,093

For other languages, we use GHTorrent to extract data from all GitHub projects before the first bug in their associated benchmark. Table 3.1 shows the number of projects and the number of instances in all training sets.

Selecting State-of-the-art Tools for Comparison: We compare CoCoNuT against 27 APR techniques, including all Java tools used in previous comparisons [140], two additional recent techniques [108, 210], five state-of-the-art C tools [4, 148, 150, 169], and two NMT-based techniques [33, 134]. We chose DLFix [134] and SequenceR [33] over other NMT techniques [80,176,237] for comparison, because [237] only generates templates and [80,176] focus on compilation errors, which is a different problem.

Training, Tuning, and Inference: For tuning, we pick a random sample of 20,000 instances as our validation dataset and use the rest for training.

We use random search to tune hyperparameters. We limit the search space to reasonable values: embedding size (50-500), convolutional layer dimensions (128*(1-5), (1-10)), number of convolutional layers (1-10), dropout, gradient clipping level, and learning rate (0-1). For tuning, we train 100 models for one epoch on the 2006 Java training set with different hyperparameters and rank the hyperparameters sets based on their perplexity [92], which is a standard metric in NLP that measures how well a model generates a sequence. We train the top-k (default k=10) models using ReduceLROnPlateau schedule, with a plateau of 10^{-4}, and stop at convergence or until we reach 20 epochs. In inference mode, we use beam search with a beam width of 1,000.

Infrastructure: We use the Pytorch [194] implementations of LSTM, Transformer, and FConv provided by fairseq-py [2]. We train and evaluate CoCoNuT on three 56-core servers with NVIDIA TITAN V, Xp, and 2080 Ti GPUs.

3.5 Evaluation and Results

Realistic Evaluation Setup: To evaluate CoCoNuT, we use six benchmarks commonly used for APR that contain realistic bugs.

When dealing with time-ordered data, it is not uncommon to incorrectly set up the evaluation [230]. If the historical data used to build the APR technique is more recent than the bugs in the benchmarks, it could contain helpful information (e.g., code clones and regression bugs) that would realistically be unavailable at the time of the fix. Using training/validation instances that are newer than the bugs in the benchmark to train or validate our models would be an incorrect setup and might artificially improve the performances of the models.

This incorrect setup potentially affects all previous APR techniques that use historical data. Although the effect may be smaller, even pattern-based techniques could suffer from this problem since patterns might have been manually learned from bugs that were fixed after the bugs in the benchmarks. To the best of our knowledge, we are the first to acknowledge and address this incorrect setup issue in the context of APR. The using-future-data threat is also one of the reasons that we did not use k-fold cross-validation for evaluation.

To address the setup issue, we extract the timestamp of the oldest bug in the benchmark (based on the date of the fixing commits) and only use training and validation instances before that timestamp. However, adding newer data to the training set would help CoCoNuT fix more recent bugs (e.g., using data until 2010 would be helpful to fix bugs from 2011). A straightforward solution would be to retrain CoCoNuT using different training data for each bug timestamp in the benchmark; however, this is not scalable. Instead, we split our benchmark into two parts. The first part contains bugs from 2006 to 2010 and is used to evaluate CoCoNuT trained with data from before 2006. The second part of the benchmark contains bugs from 2011 to 2016 and is used to evaluate CoCoNuT trained with data from before 2011 (including data from before 2006). This split allows CoCoNuT to learn from instances up to 2010 to fix newer bugs while keeping the overhead reasonable. We then combine the results of CoCoNuT on these two sub-benchmarks to obtain the final number of bugs fixed. With this correct setting, CoCoNuT has no access to data that would be unavailable in a realistic scenario.

Similar to previous work [32, 87, 94, 115, 127, 138, 139, 140, 141, 143, 167, 200, 201, 209, 210, 252, 259, 260], we stop CoCoNuT after the first generated patch that is successfully validated against the test suite. If no patch passes the test suite after the limit of six hours, we stop and consider the bug not repaired by CoCoNuT. For evaluation purposes only,

three co-authors manually compare the plausible patches (i.e., patches that pass the test cases) to the developers' patches and consider a patch *correct* if they all agree it is identical or semantically equivalent to the developers' patch using the equivalence rules described in previous work [142].

3.5.1 RQ1: How does CoCoNuT perform against state-of-the-art APR techniques?

Approach: Table 3.2 shows that we compare CoCoNuT with 27 state-of-the-art G&V approaches on six benchmarks for four different programming languages. We use `Defects4J` [101] and `QuixBugs` [135] for Java, `Codeflaws` [231] and `ManyBugs` [126] for C, and `QuixBugs` [135] for Python. For JavaScript; we use the 12 examples associated with common bug patterns in JavaScript described in previous work (`BugAID`) [82]. The total number of bugs in each dataset is under the name of the benchmark.

We compare our results with popular (e.g., GenProg) and recent (e.g., TBar) G&V techniques for C and Java. We do not compare with the JavaScript APR technique Vejovis [191] since it only fixes bugs in Document Object Model (DOM). We extract the results for each technique from a recent evaluation [138] and cross-check against original results when available. Perfect buggy location results are extracted from the GitHub repository of previous work [142] and manually verified to remove duplicate bugs.

We run Angelix, Prophet, SPR, and GenProg on the entire `Codeflaws` dataset because these techniques had not been evaluated on the full dataset. We use the default timeout values provided by the `Codeflaws` dataset authors. Since `Codeflaws` consists of small single-file programs, it is unlikely that these techniques would perform differently with a larger timeout.

The results in Table 3.2 are displayed as x/y, with x the number of correct patches that are ranked first by an APR technique, and y the number of plausible patches. We also show in parentheses the number of bugs fixed by CoCoNuT that have not been fixed by other techniques. '-' indicates that a technique has not been evaluated on the benchmark or does not support the programming language of the benchmark. For the `BugAID` and `ManyBugs` benchmarks, we could not run the validation step; therefore we only display the number of patches that are identical to developers' patches († in the Table) and cannot show the number of plausible patches.

Since different approaches used different fault localization (FL) techniques, we separate them based on FL types (Column FL), as was done in previous work [139]. Standard

Table 3.2: Comparison with state-of-the-art G&V approaches. The number of bugs that only CoCoNuT fixes is in parentheses **(307).** The results are displayed as x/y, with x the number of bugs correctly fixed. and y the number of bugs with plausible patches. * indicates tools whose manual fix patterns are used by TBAR. Numbers are extracted from either the original papers or from previous work [142] which reran some approaches with perfect localization. † indicates the number of patches that are identical to developer patches—the minimal number of correct patches. - indicates tools that have not been evaluated on a specific benchmark. The highest number of correct patches for each benchmark is in bold.

FL	Tool	Java		C		Python	JavaScript
		Defects4J 393 bugs	QuixBugs 40 bugs	Codeflaws 3,902 bugs	ManyBugs 69 bugs	QuixBugs 40 bugs	BugAID 12 bugs
Standard	**1 Angelix** [169]	-	-	318/591	**18/39**	-	-
	2 Prophet [148]	-	-	301/839	15/39	-	-
	3 SPR [150]	-	-	283/783	11/38	-	-
	4 Astor* [167]	-	6/11	-	-	-	-
	5 LSRepair [141]	19/37	-	-	-	-	-
	6 DLFix [134]	29/65	-	-	-	-	-
Supplemented	**7 JAID** [32]	9/31	-	-	-	-	-
	8 HD-Repair* [125]	13/23	-	-	-	-	-
	9 SketchFix* [87]	19/26	-	-	-	-	-
	10 ssFix* [259]	20/60	-	-	-	-	-
	11 CapGen* [252]	21/25	-	-	-	-	-
	12 ConFix [108]	22/92	-	-	-	-	-
	13 Elixir* [209]	26/41	-	-	-	-	-
	14 Hercules [210]	49/72	-	-	-	-	-
Perfect	**15 SOSRepair** [4]	-	-	-	16/23	-	-
	16 Nopol [261]	2/9	1/4	-	-	-	-
	17 (j)Kali [167, 201]	2/8	1/2	-	3/27	-	-
	18 (j)GenProg [127, 167]	6/16	0/2	[255-369]/1423	2/18	-	-
	19 RSRepair [200]	10/24	2/4	-	2/10	-	-
	20 ARJA [269]	12/36	-	-	-	-	-
	21 SequenceR [33]	12/19	-	-	-	-	-
	22 ACS [260]	16/21	-	-	-	-	-
	23 SimFix* [94]	27/50	-	-	-	-	-
	24 kPAR* [138]	29/56	-	-	-	-	-
	25 AVATAR* [139]	29/50	-	-	-	-	-
	26 FixMiner* [115]	34/62	-	-	-	-	-
	27 TBar [140]	**52/85**	-	-	-	-	-
	CoCoNuT (not fixed by others)	44/85 **(6)**	**13/ 20 (10)**	**407/ 576 (269)**	7 †/- **(0)**	**19/21 (19)**	**3** †/- **(3)**
	Total bugs fixed by CoCoNuT			**493 (307)**			

FL-based approaches use a traditional spectrum-based fault localization technique. Supplemented FL-based APR techniques use additional methods or assumptions to improve FL. For example, HD-Repair assumes that the buggy file and method are known. Finally, Perfect FL-based techniques assume that the perfect localization of the bug is known. According to recent work [138, 142], this is the preferred way to evaluate G&V approaches, as it enables fair assessment of APR techniques independently of the fault localization approach used.

We exclude iFixR [116] because it uses kPAR to generate fixes which is already in the Table. We choose kPAR since its evaluation is similar to our evaluation setup and allows for a fairer comparison. iFixR proposes to use bug reports instead of test cases. However, we believe that it is reasonable to keep test cases for validation because, even if they were committed at a later stage, they were often available to the developers at the time of the bug report since developers generally discover bugs by seeing a program fail given one failing test case. Regardless, this is still a fair comparison with the 27 existing techniques which all use test cases for validation.

TBar is a recent pattern-based technique that uses patterns implemented by previous work (techniques marked with a * in Table 3.2). *TBar shows how a combination of most existing pattern-based techniques behaves with perfect localization.*

Results: Overall, Table 3.2 shows that CoCoNuT fixes 493 bugs across six bug benchmarks in four programming languages. CoCoNuT is the best technique on four of the six benchmarks and is the only technique that fixes bugs in Python and JavaScript, indicating that CoCoNuT is easily portable to different programming languages with little manual effort.

On the **Java** benchmarks, CoCoNuT outperforms existing tools on the `QuixBugs` benchmark, fixing 13 bugs, including 10 bugs that have not been fixed before. On `Defects4J`, CoCoNuT performs better than all techniques but TBar and Hercules. In addition, 6 bugs CoCoNuT fixes have not been fixed by any other techniques. Two of the 6 bugs require new fix patterns that have not been found by any previous work from a decade of APR research. We investigate these six bugs in detail in RQ2. In addition, fifteen of the bugs fixed by Hercules are multi-hunk bugs (i.e., bugs that need changes in multiple locations to be fixed), currently out of scope for CoCoNuT which mostly generates single-hunk bug fixes. In the future, the method used by Hercules to fix multi-hunk bugs could be applied to CoCoNuT.

Two of the 240 `Defects4J` bugs from after 2010 can only be fixed by models trained with the training set containing data up to 2010. Surprisingly, the models trained with data from before 2006 stay relevant and can fix bugs introduced 4 to 10 years after the

training data. This highlights the fact that, while training DL models is expensive, such models stay relevant for a long time and do not need expensive retraining.

In `Defects4J`, 43% (19 out of 44) of the bugs fixed by CoCoNuT are not fixed by TBar (the best technique), hence CoCoNuT complements TBar very well. There are also several bugs fixed by TBar that CoCoNuT fails to fix. Seven of them are if statement insertions (e.g., null check), and three of them are fixes that move a statement to a different location. These bugs are difficult to fix for CoCoNuT because we targeted bugs that are fixed by modifying a statement, hence these two transformations do not appear in our training set. Fixing bugs by inserting if statements has been widely studied [140, 150, 260, 261]. Instead, we propose a new technique that fixes bugs that are more challenging for other techniques to fix.

While we generate 1,000 patches for each bug per model, correct patches are generally ranked very high. Ten of the 44 correct patches in `Defects4J` are ranked first by one of the models. The average rank of a correct patch is 63 and the median rank is 4. The worst rank of a correct patch is 728.

A potential threat to validity is that some tools use different fault localization techniques. While we cannot re-implement all existing tools using perfect localization (e.g., some tools are not publicly available), we try our best to mitigate this threat. First, we consider 13 tools that also use perfect localization for comparison, including TBar, which fixed the most number of bugs in `Defects4J`. Second, TBar uses fix patterns from 10 tools (marked with * in Table 3.2) with perfect localization. Thus, although some of these 10 tools do not use perfect fault localization, we indirectly compare with these tools using perfect fault localization. CoCoNuT fixes 6 bugs that have not been fixed by existing tools including TBar.

On the **C** benchmarks, CoCoNuT fixes 414 bugs, 269 of which have not been fixed before. CoCoNuT outperforms existing techniques on the `Codeflaws` dataset, fixing 105 more bugs than Angelix and has a 59% precision (407 out of 576). On the `ManyBugs` benchmark, CoCoNuT fixes seven bugs, outperforming GenProg, Kali, and RSRepair but is outperformed by other techniques.

We manually check for semantic equivalence for all generated patches except for GenProg on `ManyBugs`. Instead, we manually check a random sample of 310 out of the 1,423 plausible patches for `Codeflaws` generated by GenProg, because GenProg rewrites the program before patching it, e.g., replacing a `for` loop with a `while` loop, which makes it difficult to manually investigate all 1,423 plausible patches. The margin of error for this sample is 4% with a 95% confidence level, thus, Table 3.2 shows the projected range of the number of `Codeflaws` bugs that GenProg fixes.

```
-   static float toJavaVersionInt(String version) {
+   static int toJavaVersionInt(String version) {
```

Figure 3.4: CoCoNuT's Java Patch for Lang 29 in `Defects4J` that have not been fixed by other techniques.

On the **JavaScript** and **Python** bug benchmarks, CoCoNuT fixes three and 19 bugs respectively.

Since different bug benchmarks contain different distributions of different types of bugs, and different APR tools fix different types of bugs, it is important to evaluate new APR techniques on different benchmarks for a fair and comprehensive comparison. CoCoNuT is the best technique on four of the six benchmarks.

Summary: CoCoNuT is the first approach that has been successfully applied without major re-implementation to different programming languages, fixing **493 bugs** in six benchmarks for four popular programming languages, **307** of which that have not been fixed by existing work, including six in `Defects4J`, a dataset that has been heavily used to evaluate 22 other tools.

3.5.2 RQ2: Which bugs only CoCoNuT can fix?

Approach: For the bugs only CoCoNuT can fix in `Defects4J`, we rerun TBar, the best technique for Java, on our hardware, with perfect localization, and without a time limit to confirm that TBar cannot fix these bugs, even under the best possible conditions. For C, as stated in RQ1, we run Angelix, SPR, Prophet, and GenProg on `Codeflaws` with the same hardware we used for CoCoNuT for a fair comparison. CoCoNuT is the only technique fixing bugs in the Python and JavaScript benchmarks.

Results: CoCoNuT fixes bugs by automatically learning new patterns that have not yet been discovered, despite a decade of research on Java fix patterns. Mockito 5 (Figure 3.1a) is a bug in `Defects4J` only CoCoNuT fixes because it requires patterns that are not covered by existing pattern-based techniques (i.e., exception type update). Lang 29 (Figure 3.4) is another bug that cannot be fixed by other tools because existing pattern-based techniques such as TBar do not have a pattern for updating the return type of a function. TBar cannot generate any candidate patches for these two bugs.

Thanks to the context-aware architecture, CoCoNuT fixes bugs other techniques do not fix by correctly extracting donor code (e.g., correct variable names) from the context, while

```
- int indexOfDot=namespace.indexOf('.');
+ int indexOfDot=namespace.lastIndexOf('.');
```

(a) Java patch for Closure 92 in `Defects4J`.

```
- int end = message.indexOf(templateEnd,start)
   ;
+ int end = message.lastIndexOf(templateEnd,
   start);
```

(b) Similar change to Closure 92 bug occurring in the training data.

Figure 3.5: Example of patches fixed only by CoCoNuT and related changes in the training set.

```
- while True:
+ while queue:
```

(a) Python patch for BREADTH_FIRST_SEARCH in `QuixBugs`.

```
- while (true) {
+ while(!queue.isEmpty()) {
```

(b) Java patch for BREADTH_FIRST_SEARCH in `QuixBugs`.

Figure 3.6: BREADTH_FIRST_SEARCH bugs fixed by CoCoNuT in both Python and Java `QuixBugs` benchmarks.

techniques such as TBar rely on heuristics. An example of such bugs is in Figure 3.1b as explained in Section 3.1.

CoCoNuT is the only technique that fixes Closure 92 because it learns from historical data (Figure 3.5b) that the `lastIndexOf` method in the `String` library is a likely candidate to replace the `indexOf` method. Other techniques such as TBar fail to generate a correct patch because the donor code is not in their search space.

CoCoNuT directly learns patterns from historical data without any additional manual work, making it portable to different programming languages. Figure 3.6 shows the **same** bug in both the Java (Figure 3.6a) and Python (Figure 3.6b) implementations of the `QuixBugs` dataset. CoCoNuT fixes both bugs, even if the patches in Java and Python are quite different. Adapting pattern-based techniques for Java to Python would require much work because the fix-patterns are very different, even for the same bug.

Summary: CoCoNuT fixes **307** bugs that have not been fixed by existing techniques, including six bugs in `Defects4J`. CoCoNuT fixes new bugs by (1) automatically learning **new patterns** that have not been found by previous work, (2) extracting donor code **from the context of the bug**, and (3) extracting donor code **from the historical training data**.

3.5.3 RQ3: What are the contributions of the different components of CoCoNuT and how does it compare to other NMT-based APR techniques?

Approach: To understand the impact of each component of CoCoNuT, we investigate them individually. More specifically, we focus on three key contributions: (1) the performance of our new NMT architecture compared to state-of-the-art NMT architectures, (2) the impact of context, and (3) the impact of ensemble learning. We compare CoCoNuT with DLFix [134] and SequenceR [33], two state-of-the-art NMT-based APR techniques and three other state-of-the-art NMT architectures (i.e., LSTM [153], Transformer [242], and FConv [71]). These models have not been used for program repair, so we implemented them in the same framework as our work (i.e., using Pytorch [194] and the fairseq-py [2] library). To ensure a fair comparison, we tune and train the LSTM, Transformer, and FConv similarly to CoCoNuT. SequenceR [33] uses an LSTM encoder-decoder approach to repair bugs automatically. We use the numbers reported by SequenceR and DLFix's authors on the `Defects4J` dataset for comparison since working versions of SequenceR and DLFix were unavailable at the time of writing. DLFix [134] uses a new treeRNN architecture that represents source code as a tree.

Comparison with State-of-the-art NMT: CoCoNuT fixes the most number of bugs, with 44 bugs fixed in `Defects4J`. DLFix is the second best, fixing 29 bugs, followed by FConv with 21 bugs while Transformer and SequenceR have similar performances, fixing 13 and 12 bugs respectively. The last baseline, LSTM, performs poorly with only 5 bugs fixed. These results demonstrate that CoCoNuT performs better than state-of-the-art DL approaches and that directly applying out-of-the-box deep-learning techniques for APR is ineffective.

Impact of the Context: Figure 3.7 shows the total number of bugs fixed using the top-k models, with k from 1 to 10. For all k, CoCoNuT outperforms an ensemble of FConv models trained without using context. Our default CoCoNuT (with $k = 10$ models) fixes six more bugs than using models without context (44 versus 38).

Advantage of Ensemble Learning: Figure 3.7 shows that as k increases from 1 to 10, the number of bugs that CoCoNuT fixes increases from 22 to 44, a 50% improvement. We observe a similar trend for the FConv models, indicating that ensemble learning is beneficial independently of the architecture used. Model 9 and Model 7 are the best FConv and Context-aware models respectively, both fixing 26 bugs.

While increasing k also increases the runtime of the technique, this cost is not prohibitive because CoCoNuT is an offline technique and can be run overnight. In the worse case, with k=10, CoCoNuT takes an average of 16.7 minutes to generate 20,000 patches for one bug (for k=1, it takes an average of 1.8 minutes per bug).

While CoCoNuT fixes 44 in `Defects4J`, the average number of bug fixed by a single model is 15.65. in `Defects4J`, 40% of the bugs are fixed by five or fewer models. Six of the correctly fixed bugs are only fixed by one model, while only two bugs are fixed by all models. This indicates that different models in our ensemble approach specialize in fixing different bugs.

Summary: The new NMT architecture we propose performs significantly better than baseline architectures. In addition, using ensemble learning to combine the models improves the results, with a 50% improvement for k=10 compared to k=1.

3.5.4 RQ4: Can we explain why CoCoNuT can (or fail to) generate specific fixes?

The Majority of the Fixes are not Clones: By learning from historical data, the depth of our neural network allows CoCoNuT to fix complex bugs, including the ones that require

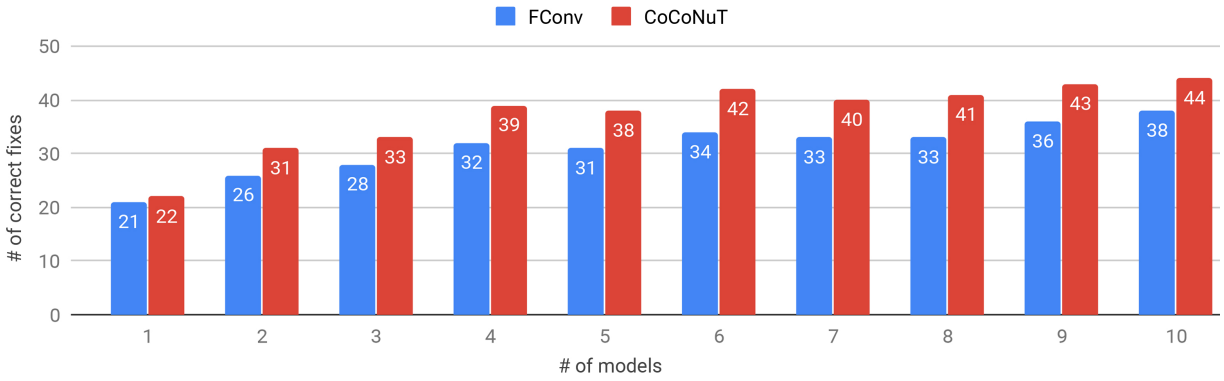


Figure 3.7: Number of bugs fixed as the number of models considered in ensemble learning increases.

```
- Calendar c = new GregorianCalendar(mTimeZone);
+ Calendar c = new GregorianCalendar(mTimeZone, mLocale);
```

Figure 3.8: CoCoNuT's Java patch for Lang 26 in `Defects4J`.

generating new variables. As discussed at the start of Section 3.5, we only keep training and validation instances from before the first bugs in our benchmarks for a fair evaluation. As a result, the exact same bug cannot appear in our training/validation sets and evaluation benchmarks. However, the same patch may still be used to fix different bugs introduced at different times in different locations. Having such patch clones in both training and test sets is valid, as recurring fixes are common. The majority of the bugs fixed by CoCoNuT do not appear in the training sets: only two patches from the C benchmark and one from the JavaScript benchmark appear in the training or validation sets. This indicates that CoCoNuT is effective in learning and generating completely different fixes.

Analyzing the Attention Map: CoCoNuT can also fix bugs that require complex changes. For example, the fix for *Lang 26* from the `Defects4J` dataset shown in Figure 3.8 requires injecting a new variable `mLocale`. This new variable only appears four times in our training set, and never in a similar context. However, `mLocale` contains the token `Locale` which co-occurs in our training set with the buggy statement tokens `Gregorian`, `Time`, and `Zone`.

The attention map in Figure 3.9 confirms that the token `Time` is important for generating the `Locale` variable. Specifically, the tokenized input is shown on the y-axis while the tokenized generated output is displayed on the x-axis. The token `<CAMEL>` between `m` and

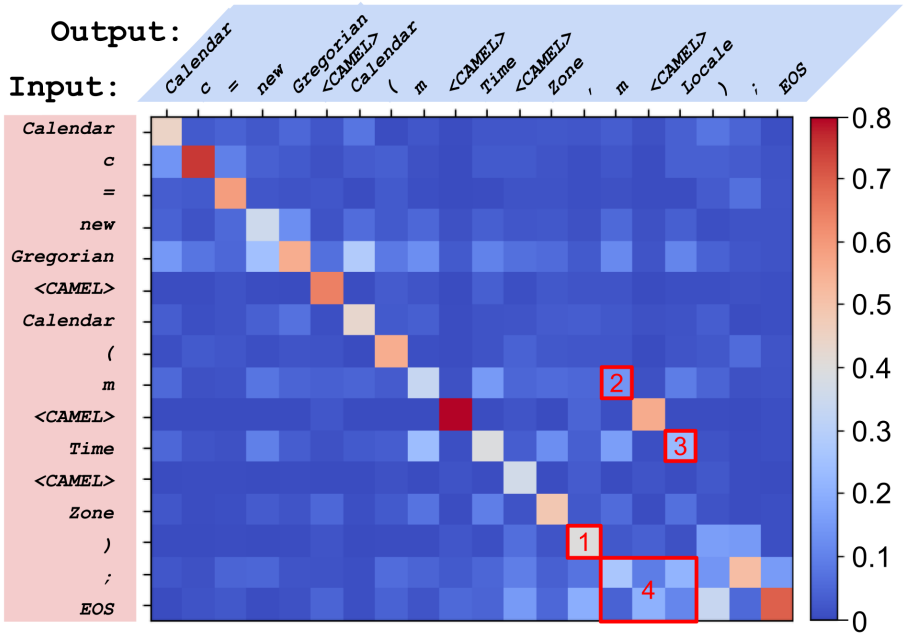


Figure 3.9: Attention map for the correct patch of *Lang 26* from the `Defects4J` benchmark generated by CoCoNuT.

`Locale` indicates that these two tokens form one unique variable. The attention map shows the relationship between the input tokens (vertical axis) and the generated tokens (horizontal axis). The color in a cell represents the relationship of corresponding input and output tokens. The color scale, shown on the right of the figure, varies from 0 (dark blue) to 0.6 (dark red) and indicates the contribution of each input token for generating an output token.

This attention map helps us understand why the model generates specific tokens. For example, the second `m` in the output is generated because of the token `m` in the input (part of `mTimeZone`), showing that the network can keep the naming convention when generating new variables (square labeled 2 in Figure 3.9). The token `Locale` is mostly generated because of the token `Time`, indicating that the network is confident these tokens should be together (square 3). Finally, the tokens forming the variable `mLocale` are all influenced by the input tokens `) ;` and `EOS`, indicating that this token is often used right before the end of a statement (i.e., as the last parameter of the function call, rectangle 4 on Figure 3.9). This example shows how the attention map can be used to understand why CoCoNuT generates a specific patch.

Limitations of Test Suites: Patches that pass the test suite but are incorrect are an issue that CoCoNuT and other APR techniques share. CoCoNuT could generate a correct fix for 8 additional bugs if more time is given; however, for these 8 bugs, CoCoNuT generates an incorrect patch that passes the test suite (we only validate the first candidate patch due to time considerations) or timed out before the correct fix. Using enhanced test suites proposed in previous work [263] may alleviate this issue.

3.5.5 Execution Time

Data Extraction: Extracting data from open-source repositories and training CoCoNuT are one-time costs that can be amortized across many bugs and should not be counted in the end-to-end time to fix one bug. For Java, extracting data from 59,237 projects takes five days using three servers.

Training Time: The median time to train our context-aware NMT model for 1 epoch during tuning is 8.7 hours. On average, training a model for 20 epochs takes 175 hours on one GPU. Transformers and FConv networks are faster to train, taking an average of 2.5 and 2.7 hours per epoch. However, training the LSTM network is much slower (22 hours per epoch).

This one time cost is to be compared to **the decade of research** spent designing and implementing new fix patterns.

Cost to fix one bug: Once the model is trained, the main cost is the inference (i.e., generating patches) and the validation (i.e., running the test suite). During inference, generating 20,000 patches for one bug (CoCoNuT default setup) takes 16.7 min on average using one GPU. On our hardware, CoCoNuT's median execution time to validate a bug is six sec on `Codeflaws` and 6 min on `Defects4J` (benchmark with the largest programs and test suites).

CoCoNuT median **end-to-end time** to fix a bug varies from 16.7 min on `Codeflaws` to 22.7 min on `Defects4J`. In comparison, on identical hardware, the median time of other tools that we ran on `Codeflaws` (Angelix, GenProg, SPR, and Prophet) varies from 30 sec to 4 min. TBar, on the same hardware, has an 8 min median execution time on `Defects4J`.

While the end-to-end approach of CoCoNuT is slower than existing approaches, it is still reasonable. In addition, we can shorten the execution time by reducing the number of patches generated in inference. Generating only 1,000 patches (i.e., 50 patches per model) would reduce CoCoNuT's end-to-end time to 2 min on `Codeflaws` and 7 min on `Defects4J`

while still fixing most of the bugs (e.g., 34 in `Defects4J`). Parallelism on multiple GPUs and CPUs can also speed up the inference.

3.6 Threats to Validity

The dataset used to train our approach is different from the ones used by other work (e.g., SequenceR and DLFix), which could impact our comparison results. However, the choice of datasets and how to extract and represent data are a key component of a technique. In addition, we compare our approach with LSTM, FConv, and Transformer architectures using the same training data and hardware, which shows that CoCoNuT outperforms all other three. Finally, both DLFix and SequenceR use data that was unavailable at the time of the bug fix (i.e., their models are trained using instances committed after the bugs in Defects4J were fixed), which could produce false good results as shown by prior work [230].

A challenge of deep learning is to explain the output of a neural network. Fortunately, for developers, the repaired program that compiles and passes test cases should be self-explanatory. For users who build and improve CoCoNuT models, we leverage the recent multi-step attention mechanism [71] to explain why a fix was generated or not.

There is a threat that our approach might not be generalizable to fixing bugs outside of the tested benchmarks. We use six different benchmarks in four different programming languages to address this issue. In the future, it is possible to evaluate our approach in additional benchmarks [81, 160, 208].

There is randomness in the training process of deep-learning models. We perform multiple runs and find that the randomness in training has little impact on the performances of the trained models.

3.7 Summary

We propose CoCoNuT, a new end-to-end approach using NMT and ensemble learning to automatically repair bugs in multiple languages. We evaluate CoCoNuT on six benchmarks in four different programming languages and find that CoCoNuT can repair 493 bugs including 307 that have not been fixed before by existing techniques. In the future, we plan to improve our approach to work on multi-hunk bugs.

Chapter 4

On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files

4.1 Motivation

Many different applications exist for displaying a given type of electronic document, and inconsistencies between these applications can be critical sources of miscommunication. For example, there are many Portable Document Format (PDF) readers (such as Acrobat Reader, Evince, and Firefox), image file readers (such as ACDSee, Eye of GNOME, and Geeqie), and word document readers (such as Microsoft Word, Abiword, and Libreoffice). Electronic documents are increasingly displacing paper documents for delivering important information including medical advice, bills, maps, and tax information. To avoid miscommunication, it is crucial to display an electronic file consistently across different file readers.

The PDF format was created to alleviate the portability problems of electronic files. Unfortunately, there are still many inconsistencies among PDF file readers. For example, the Chrome and Mozilla support forums contain hundreds of complaints from users about PDF files being displayed differently across readers. These issues include drug information sent to doctors that cannot be properly displayed or opened [40], customers unable to read their online bills [183], and web designers worrying that customers cannot correctly display

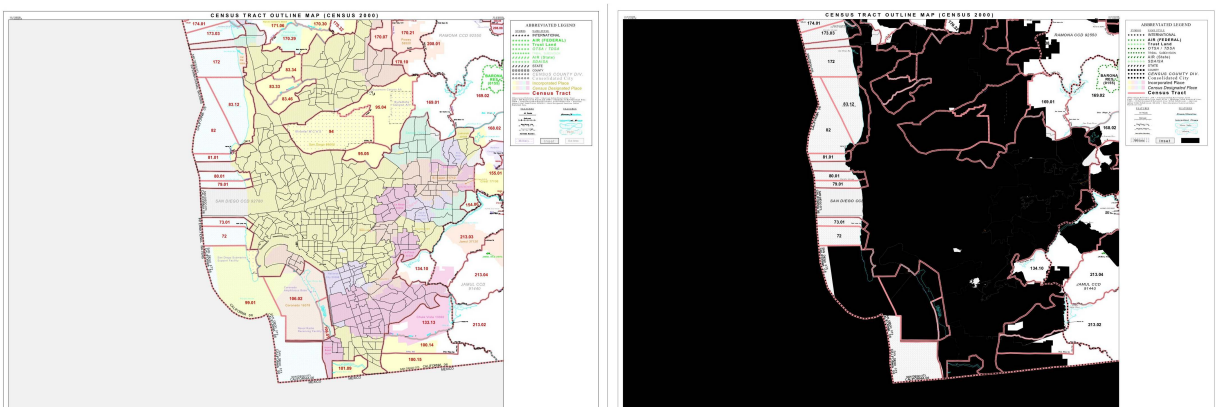


Figure 4.1: An example of a bug in `Firefox`. `Chromium` rendering on the left, `Firefox` rendering on the right.

the PDF files on their websites [39]. There are two main causes of inconsistencies between PDF readers.[1]

(1) Bugs in readers: PDF readers, such as `Acrobat Reader`, `Evince`, and `Chromium`, contain bugs. Figure 4.1 presents such a bug in `Firefox`'s embedded PDF reader. The image on the left shows the rendering of a PDF by `Chromium`, while the image on the right shows the same document rendered by `Firefox`. `Firefox` fails to display the map properly and fills large areas with black colour. `Firefox` developers confirmed our bug report [27].

(2) Bugs in files: PDF files contain bugs, causing readers to display them inconsistently, or even fail to load them. For example, if a special font is not embedded in a PDF file, some readers fail to display the contents of this file on a computer that does not contain this font. If one can automatically detect those inconsistencies, the creators or owners of PDF files could modify them to ensure the files they share will be displayed as intended by all users.

There is a blurry line between bugs in readers and bugs in files, as some PDF readers are more tolerant to errors than others. In addition, we learned that in many cases, developers are willing to provide workaround patches to the readers to tolerate bugs in malformed files. For example, we reported a damaged file to the `Poppler` rendering library. In the comments posted in the bug tracker [26] the developers confirmed that the file was broken, but since key parts (e.g., the object catalogue) were undamaged, they decided that `Poppler` should be able to render this file and proposed a fix.

[1] In this work, we use the term *PDF reader*, or just *reader*, to refer to any application that takes a PDF file as input, such as PDF viewers, editors and processing utilities.

Regardless of whether the bugs are in readers or files, they cause content to be displayed inconsistently, affecting the correct communication of information. As a result, it is important to design techniques to automatically detect *cross-reader inconsistencies*, which could reveal such critical bugs affecting electronic documents and can also lead to better document recovery techniques, recently tackled by the research community [47, 117, 149].

In this work, we conduct a large-scale empirical study to understand the severity of cross-reader inconsistencies, and propose new techniques to automatically detect them. Specifically, we study the inconsistent display of popular PDF readers on more than 230,000 real-world PDF files previously mined from US government websites [68]. Since those files are offered by the government, it is essential that they are opened and rendered consistently by a wide range of PDF readers.

By manually inspecting how different readers behave on a small fraction of these PDF files, we identify several challenges of automatically detecting bugs in readers and files. First, a pixel-by-pixel comparison of the rendered images is too strict, leading to many spurious alarms, e.g., due to tiny differences related to how certain characters are displayed. Second, capturing and comparing rendered images accurately is expensive and impractical for such a large data set, even on a cluster of machines. Third, many inconsistencies are caused by the same underlying error in a reader or PDF file, resulting in many redundant inconsistencies to be detected.

To address these challenges, we have devised a multi-stage approach for automatically detecting crashes and inconsistencies. First, we load every document in a portfolio of readers, and (1) report any crashes, and (2) record all warnings and errors logged by the readers. Second, for each reader, we cluster documents based on the warnings and errors issued in the first phase. Third, we select a representative from each cluster, load it in several readers, and then capture and compare the rendered images across readers. Then, we use a form of delta debugging [271] to precisely localize the source of inconsistency and generate a reduced PDF file that only displays a small number of inconsistent objects. Finally, we analyze any detected inconsistencies and report them to developers.

We apply our approach to the 230K files in the `Govdocs1` database, which automatically detected 30 unique bugs in popular PDF readers such as the ones embedded by Chromium and Firefox. We also reported 33 bugs (including 11 manually detected on Windows readers), some of which had already been confirmed or fixed by developers. These bugs affect the correctness of the displayed PDF file, causing e.g., readers to crash or PDF elements to be skipped.

In summary, we make the following contributions:

- We conduct the first (to the best of our knowledge) study of cross-reader inconsistencies by manually examining and categorising a random sample of 2,313 PDF files from the `Govdocs1` database. We found that cross-reader inconsistencies are common—314 out of 2,313 (13.5%) files are displayed inconsistently—the displayed image is different in at least one reader. Common symptoms include missing images and font inconsistencies.
- We design a technique to automatically detect crashes and inconsistencies across PDF readers based on error filtering, clustering, and image processing techniques.
- We apply our techniques to the 230K files in the `Govdocs1` database to find cross-reader inconsistencies, which detected 30 unique bugs in popular PDF readers.
- We develop a technique to precisely locate document elements causing an inconsistency. This can help developers identify potential bugs in document readers faster.
- We assemble a database of documents that expose errors in these readers, annotated with the types of errors exposed, error messages triggered and a link to the discussion of each bug report. This database could help researchers and practitioners address other software reliability challenges including testing other readers, and diagnosing and fixing bugs in readers and PDF files.

Our database is made available at http://srg.doc.ic.ac.uk/projects/pdf-errors.

4.2 Approach

Our technique takes a set of PDF documents and a portfolio of PDF readers as input, cross-checks whether these readers open the same PDF document consistently, and outputs a list of documents that crash readers or display differently in the readers. Display inconsistencies in turn indicate either bugs in the readers or in the PDF document. For each document that is detected inconsistent, we use delta debugging to produce a reduced PDF file that only displays inconsistent objects. Our technique also outputs clusters of PDF documents, each of which contains PDF documents that are likely to expose the same bug.

Figure 4.2 presents a high-level overview of our approach, which consists of three main phases: the filtering phase (Section 4.2.1), the inconsistency detection phase (Section 4.2.2) and the inconsistency localization phase (Section 4.2.3). In the filtering phase, PDF documents are opened in a portfolio of PDF readers. If a reader crashes on a PDF file, our technique reports a crash bug. Otherwise, it collects error messages and warnings emitted

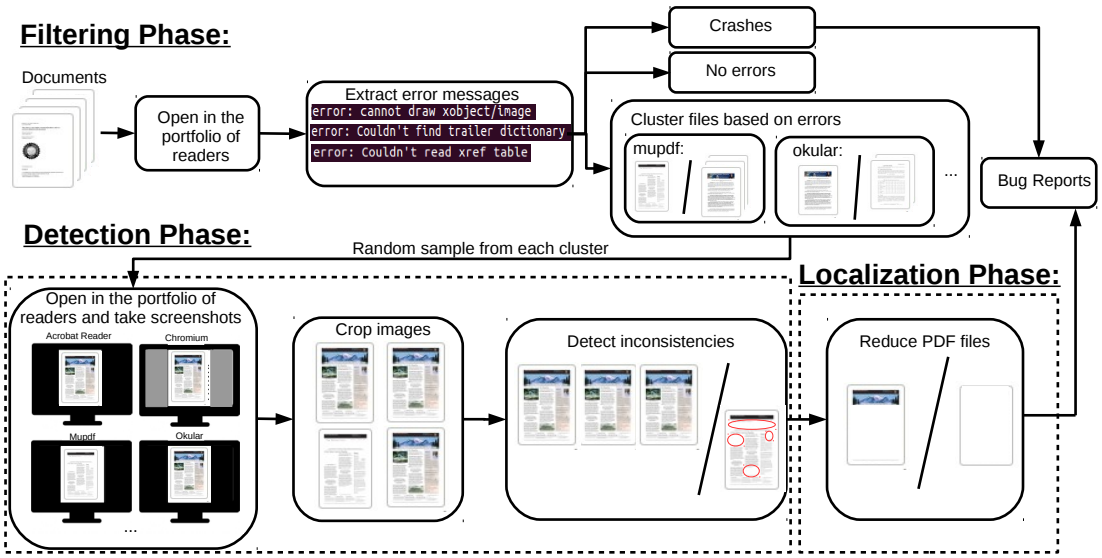


Figure 4.2: Overview of our approach, which consists of three main phases: the filtering phase (Section 4.2.1), the inconsistency detection phase (Section 4.2.2), and the inconsistency localization phase (Section 4.2.3).

by the readers to standard error, and then clusters the documents that produce similar error messages. From each cluster, our approach randomly selects a candidate and proceeds to the second phase.

In the inconsistencies detection phase, our technique opens the candidates in a set of PDF readers and captures screenshots. It then cleans the screenshots by removing some captured elements of the graphical user interface (e.g., different background colour) and applies an image similarity technique on the cleaned screenshots to detect display inconsistencies. A display inconsistency between two readers happens when parts of the same document are shown differently in the two readers.

We apply the inconsistency localization phase to each document that is displayed inconsistently. In particular, we apply delta debugging [271] to reduce the number of visible objects in the file. At the end, we obtain a reduced PDF file that only renders a small number of inconsistent objects (usually one visible object). This allows developers to know which type of objects in the PDF file is causing the issue.

We employ this three-phase approach to address the challenges described in the introduction. One challenge is that comparing rendered images is expensive, and unscalable for a large data set of 230K PDF files. For example, it takes 40 seconds to render and capture

a page with two readers and 5 seconds to compare them. Thus, the total amount of time for rendering, capturing and comparing just the first pages of the 230K documents for only two readers is approximately 125 days. To address this issue, the filtering phase employs a lightweight approach to quickly identify PDF files that are more likely to be malformed or trigger bugs in the readers. It also aims to group PDF files that cause the same bug in a reader to address the challenge of different documents exposing the same underlying bug.

Another challenge is that a pixel-by-pixel comparison between the rendered images is too strict, reporting differences that are often unnoticeable or unimportant to human eyes. Thus, in the second phase, we leverage an advanced image similarity algorithm [212] that is based on the human visual system to accurately detect bugs while tolerating unimportant differences. We also report how other image similarity algorithms compare on this task.

4.2.1 Phase 1: Document filtering

The first phase of the process serves two purposes: (1) selecting the documents that are likely to be malformed or trigger presentation problems in the readers, (2) grouping the documents that exhibit similar erroneous behaviour.

Specifically, this phase opens each document in every PDF reader considered and captures the error messages printed on the terminal. Intuitively, these error messages are observable signs of potential bugs with either the reader or the document. These observable signs are features that can help us group similar bugs, as it is reasonable to assume that similar error messages are likely to indicate similar root bugs. Our results from Section 4.5 show that our filtering approach selects files that find relatively more unique bugs than in a random sample.

We analyzed the standard error messages produced by the readers (across our data set and other studied PDF documents) and prepared a set of 501 regular expressions corresponding to the observed classes of error messages. An example of a regular expression is `(Syntax Warning|Error): Could not parse ligature component "(char|old|chart)" of "(char_[0-9] +|.+_old|bar_chart)" in parseCharName`. For each document that produces at least one error message, we try to match each message against our set of regular expressions. The matching is performed separately for each document and PDF reader. We ignore a message if it does not match any of the regular expressions. Many times the same error message is emitted by different documents. We leverage this fact to cluster documents based on the similarity of these emitted error messages.

We create feature vectors with each element of the vector corresponding to a single class of error messages (as encoded by the regular expressions). The elements in the vector have

a value of 1 if the corresponding error was matched and 0 otherwise. These feature vectors are then used by a clustering algorithm to group files which trigger similar error messages.

We use the popular *K-means clustering* algorithm [158, 159]. K-means iteratively groups a population of data points around K centroids representing the centres of the clusters. We use the *K-means++* algorithm [14] to choose the initial locations of the centroids and the sum of squares (*inertia*) as a distance metric. We configure the algorithm to cluster ten times and return the best assignment.

Establishing the right value of K—the number of clusters—is a known challenge in K-means clustering. Having too few clusters results in data points with smaller degree of similarity being grouped together, while having too many clusters poses a risk of putting similar data points in separate clusters. To tackle this problem, we use a known selection algorithm for K: the *Silhouette* [205] method. Silhouette is a coefficient describing how well each of the data points in the set fits in its cluster. It is calculated by measuring how far away a data point is from its cluster neighbours and also from the other clusters. The values of the coefficient fall into the range of [-1, 1], where -1 means a bad clustering (elements not in the right clusters), 0 means overlapping clusters and 1 means a good clustering. To select the right K, we start with value 2, perform the clustering and calculate the Silhouette coefficient. We then increase K by 1 and repeat the process. We stop when the Silhouette coefficient is at least 0.9 and we use Euclidean distance as a metric. For each reader, we generate a different set of clusters of PDF files. The rationale is that a given file may expose different errors in different readers. For example, for a file F: reader R1 may generate no error messages, while reader R2 generates one error message, and reader R3 generates a different error message. This likely indicates that F exposes no bug in reader R1, but two different bugs in R2 and R3. By clustering PDF files for each reader separately, our approach can capture such differences.

4.2.2 Phase 2: Inconsistency detection

Error messages from PDF readers alone are not enough to detect inconsistencies accurately. For example, two readers may emit identical error messages for one file, but this file may still be displayed differently by the two readers because they have different recovery strategies to address the error. Therefore, a graphical detection technique, our inconsistency detection phase, is necessary to detect inconsistent images.

This phase performs three steps: (1) capturing screenshots of the PDF files rendered by different PDF readers, (2) cropping the screenshots to remove the background and GUI

elements and (3) running an image similarity algorithm to detect inconsistent displays of the same file with different readers.

Due to the size of our data set, we only consider the first page of each document when detecting inconsistencies. We explain the rationale behind that choice in more detail in Section 4.4.1.

Capturing rendered images. A straightforward approach to capturing the images rendered by PDF viewers consists of running PDF readers in full-screen mode, displaying the result on a real monitor, and taking a screenshot. However, this simple technique has several flaws. First, the screenshot quality depends on the monitor's display size. Second, taking screenshots on a real monitor is slow, and does not scale to a large set of PDF files.

To improve the scalability of the screenshot capture process and the quality of the images, we use `Xvfb` [262], a tool that renders graphics in a buffer instead of a real monitor. Using `Xvfb`, we can set up a high screen resolution and ensure that the captured screenshots are high-quality images. In addition, because the PDF files are rendered in memory, it significantly increases the speed and scalability of the screen capture.

Cropping screenshots. As we display PDF files in full-screen mode, most of the GUI control elements (e.g., for opening a file) are hidden. However, the screenshots still contain elements that differ across PDF readers that are not related to document inconsistencies. For example, if the screen resolution is larger than the rendered page, the PDF reader will display a uniform background colour that can differ across PDF readers. Therefore, we use PDFBox to extract the page size, and crop the screenshots to keep only the part of the image containing the actual document.

Detecting display inconsistencies. Once we obtain cropped screenshots for each document and PDF reader pair, we apply image processing techniques to detect display inconsistencies.

We use a state-of-the-art algorithm for image similarity detection, called *Complex Wavelet Structural Similarity Index (CW-SSIM)* [212]. This algorithm aims to detect elements of an image that appear similar to the human eye by focusing on the structure of the objects in the image. This is important because external sources can add noise (e.g., interpolation or anti-aliasing of the software and libraries used for capturing and cropping screenshots.) The CW-SSIM index is robust to such nonstructural transformations.

The goal of this similarity metric is to decompose images into families of wavelets, (i.e., visual channels) that are similar to the decomposition done by human eyes to recognize patterns [224]. Then the CW-SSIM index between two images is measured by comparing discrete values of the wavelets obtained for the two images.

The CW-SSIM index ranges from 0 to 1. A value close to 1 indicates that the two compared images are similar, hence no inconsistencies are detected. A low CW-SSIM value indicates visual inconsistencies. For n readers, we select a base reader (`Acrobat Reader`), and compare all remaining $n-1$ readers with this base reader to obtain $n-1$ CW-SSIM indexes per file. If one of these indexes is below a certain threshold, the file is considered *inconsistent*, i.e., the file is displayed differently in at least one PDF reader. If all the CW-SSIM indexes are above the threshold, the file is considered *consistent* across all the tested readers. We empirically chose a threshold of 0.88 by running a preliminary study on a small set of PDF files.

We also experimented with seven other similarity algorithms and justify our choice of CW-SSIM in Section 4.5.1.

4.2.3 Phase 3: Inconsistency localization

Once inconsistent files have been detected, we attempt to automatically find which elements of the PDF file are causing the inconsistency. To locate the inconsistent elements, we leverage a debugging technique called delta debugging [271]. The delta debugging algorithm we used is described in Algorithm 1. T is the similarity threshold and sim the similarity algorithm we used to compare screenshots (i.e., `CW-SSIM`).

The main idea is to iteratively remove visible objects of an inconsistent PDF file to generate a minimum valid PDF file only containing a small number of inconsistent objects. At each stage, we generate two PDF files, one containing half of the visible objects, and the other containing the other half. Then, we run the inconsistency detection phase on each PDF file and keep the one that still reveals the inconsistency. If both new files reveal the inconsistency, we randomly select one. This process is then repeated until one of the following three conditions is reached: (1) we obtain a PDF file containing only one inconsistent object, (2) the new PDF file is no longer inconsistent, (3) we reach the tenth iteration. Condition (2) can be reached due to false negatives in the image similarity algorithm and cases where there is no single object responsible (e.g., performance bugs). Condition (3) is to ensure that our algorithm remains scalable for PDF files containing tens of thousands of visible objects and only happened for five files.

We implement the inconsistency localization algorithm using the Apache PDFBox library, a Java library that can be used to manipulate or generate PDF files.

Algorithm 1 Delta Debugging Algorithm

Input: Inconsistent File F with objects $\{o_1,...,o_n\}$
Output: List of objects included in the minimal PDF file

Initialization: count=0
DD(F,$\{o_1,...,o_n\}$):
1: **if** ($n = 1$ **or** count $= 10$) **then**
2: **return** $\{o_1,...,o_n\}$
3: **end if**
4: F_1=F$\{o_1,...,o_{n/2}\}$
5: F_2=F$\{o_{n/2+1},...,o_n\}$
6: **if** (sim(F_1) $< T$ **and** sim(F_2) $\geq T$) **then**
7: **DD**(F_1,$\{o_1,...,o_{n/2}\}$)
8: count += 1
9: **else if** (sim(F_2) $< T$ **and** sim(F_1) $\geq T$) **then**
10: **DD**(F_2,$\{o_{n/2+1},...,o_n\}$)
11: count += 1
12: **else if** (sim(F_1) $< T$ **and** sim(F_2) $< T$) **then**
13: **random**(**DD**(F_1,$\{o_1,...,o_{n/2}\}$),**DD**(F_2,$\{o_{n/2+1},...,o_n\}$))
14: count += 1
15: **else**
16: **return** $\{o_1,...,o_n\}$
17: **end if**

Table 4.1: Basic statistics, extracted using `pdfinfo`, for the `GovDoc1` data set; `pdfinfo` failed for 15 documents.

	Min	Max	Median	Average	Total
File size	931B	68.8MB	155.1KB	579.9KB	127.8GB
Pages	1	3,200	10	27	6,443,316

4.3 Experimental setup

In this section we present the experimental setup used in our study. We describe our data set in Section 4.3.1, our portfolio of readers in Section 4.3.2 and our infrastructure in Section 4.3.3.

4.3.1 Data set

`Govdocs1` [68] is a data set of about one million documents crawled from US government web pages (the `*.gov` domain). The set contains documents and files of various formats, including PDF documents.

For this study, we extracted all the files with *.pdf extension from the data set. The total number of extracted files is 231,232. We found that amongst the extracted files there are 10 files which have the suffix `.pdf` but are not PDF files; that is 0.004% of all the files. Furthermore, we have found that there are 1,309 duplicates within the extracted files; that accounts for 0.57% of all the files. Neither the non-PDF files, nor the duplicates could have negatively influenced the results as their percentage compared to the set size is negligible.

According to the metadata information attached to the `Govdocs1` corpus, there are another 3,688 files categorized as PDF. However, since these files do not have *.pdf extension we do not consider them in our study.

In further sections of the work, when we refer to *data set* we mean the PDF files that we use, rather than the whole `Govdocs1` set, which also includes other types of files (unless we explicitly state otherwise).

The data set is appropriate for our experiments for the following reasons:

- **Widely-used publicly-available data set.** The `Govdocs1` data set has been widely used by the digital forensics community [75] and the original paper [68] discussing in detail the need for a curated corpus like `Govdocs1` has over 200 citations up to date. Also, the data set is freely available, which makes results accessible and reproducible.

- **File Content Diversity.** While it is true that `Govdocs1` files were collected from a common source (governmental web pages), the searches were performed using words randomly chosen from a dictionary, random numbers and a combination of the two. In total 25,330 unique search terms were used while crawling for the files in our PDF set. We believe that such a high number of the search terms used contributes to the data set diversity. We encountered a broad range of types of PDF documents in the set, such as forms, scanned paper documents, scientific articles, and maps.

- **File Size Diversity.** The files in the set are also diverse in terms of file size and the number of pages, as presented in Table 4.1.

- **Creating Software Diversity.** We inspected the files for optional metadata fields, *Creator* and *Producer*.[2] The *Creator* field is meant to store the name of the original software that created the document for those documents which were initially created in a different format and then converted to PDF. The *Producer* field is meant to store the name of the software that converted the documents to the PDF format. While these fields are optional, 90% of the files in our data set contain both of them.

  Using the Creator and Producer fields, we extracted the information on the operating system (OS) used to create a file. As expected, the majority of the files have been created on Windows (68%). 14% of the files were created on Unix-based systems (mostly Mac and Solaris). We could not extract OS information for the remaining files.

  We also looked at the software used to create the files. There are ∼18,000 unique creators and ∼1,400 unique producers in our PDF set. These numbers are high because they consider different versions of a product as different software. In order to perform further analysis, we grouped different versions of similar software.

  We found 54 applications that were used in the creation of at least 100 documents (either as a producer or a creator). *Adobe Acrobat Distiller* is the most popular software used by far (34% of the files were created by Distiller). This is not surprising as Distiller is the most popular way to create PDF files on Windows. The second most common tool to create PDF files in this data set is *PScript* (12%). *PScript* is a document converter used by many different programs (e.g., *Microsoft Office*, *Ghostscript*). *Acrobat PDFMaker* and the MacOS printer driver *PDFWriter* are also quite common (respectively 8% and 7%). Other commonly used software includes Microsoft Office suite (*Word*, *PowerPoint*, *Excel*), Adobe suite (*Capture*, *Photoshop*,

[2] http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf, pg. 844

Table 4.2: Numbers of documents in our data set for various versions of the PDF standard.

	1.0	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**
Files #	954	6,665	41,082	52,804	87,008	22,123	19,304	1,276
Percentage	0.4%	2.9%	17.8%	22.8%	37.6%	9.6%	8.3%	0.6%

InDesign, etc.), LaTeX converters, printer drivers (Hewlett-Packard, Canon, Lexmark, etc.), browsers (*Internet Explorer* and *Mozilla*), commercial document creator tools (*QuarkXPress*, *Corel WordPerfect*, *Amyuni*), open source document converters (*PrimoPDF*, *Ghostscript*, *OpenOffice*, *iText*), and Apple software (*Quartz*, *Keynote*).

We also found a small number of files (673) for which the Creator or Producer fields have been changed to "U.S. Gov. Printing Office" or "Government Accountability Office." It is possible that these files have been generated using software specific to the US government and therefore are not representative of PDF documents which can be found elsewhere. However, they only represent 0.1% of all the Creator and Producer fields.

Overall, our analysis of Creator and Producer metadata demonstrates the diversity of the tools used to generate the PDF files, despite the fact that all of the files have been crawled from a single source of US government domains.

- **PDF Standard Revision and Creation Date Diversity.** Table 4.2 presents a detailed breakdown of the number of files in our data set for each of the PDF standard version. There are representatives for every PDF standard revisions from 1.0 up to 1.7. The files creation date ranges from early nineties to 2009.[3] Since version 1.7, the PDF format has not changed much, with extensions regarding forms in 2008 and 2009, security in 2011 (change in the password checking algorithm), and a new format 2.0 in July 2017. Since we do not deal with password-protected files, we believe that the 2011 update has little impact on our study. While the files created with the 2017 standard could create different types of inconsistencies, this update is too recent and possibly fewer documents in this format have been created so far. The diversity of PDF versions and document creation times is a desirable feature of the data set for our study.

- **Real-world documents.** The documents from the `Govdocs1` corpus are real-world documents that have not been crafted, fuzzed or cherry-picked to highlight incorrect

[3]We filtered out all dates reported prior to the introduction of PDF 1.0 in 1992 and after the `Govdocs1` paper publication year, i.e., 2009.

behaviors or security issues of a specific PDF reader. This is important because in our study we want to find out about inconsistencies in "common" PDF files, in order to better understand how often the problem occurs in the real-world scenarios.

- **Privacy and contents issues.** Creating a new documents corpus, e.g., by crawling the web, is challenging from the legal and privacy point of view. Collected data might contain personal information that should not be redistributed, copyrighted material or illegal content, e.g., pornography. `Govdocs1` corpus was designed in a way that it can be used freely without violating the law.

4.3.2 Portfolio of readers

Many PDF readers are available for different operating systems. For this study, we focus on readers running on the Linux operating system, and evaluate 11 popular PDF readers. Three of the readers are command line utilities, but as mentioned in the introduction we use the term *reader* to denote any application that processes PDF files.

In our portfolio we have three command line tools: `pdfinfo`, `pdftk` and `ps2pdf`, six popular PDF readers: `Acrobat Reader`,[4] `Xpdf`, `Evince`, `Okular`, `qpdfview`, `MuPDF`, and two web browsers: `Firefox` and `Chromium`.

`Evince`, `Okular`, `qpdfview` and `pdfinfo` use the same PDF library, `Poppler`. However, the codebases are quite different and these readers behave differently. A good example is one of the bugs we reported[5] in which an inconsistency is visible for `Evince` but not for `Okular`. The PDF backend is only a part of the final result, as every PDF reader also needs the rendering component that shows results on the screen. Because of that, we believe it makes sense to test readers based on the same backend, even though some of the bugs are common for all of them.

Table 4.3 presents the readers that we use in each phase. In the filtering phase we do not consider the output from `Firefox`, `Chromium` and `Acrobat Reader`, because we found that these readers do not produce any useful PDF parsing messages on the terminal (of these three, only `Firefox` emitted parsing-related messages, but only for five of all the tested documents). In the inconsistency detection and localization phases we do not use `pdfinfo`, `pdftk` and `ps2pdf`, as they are command-line tools and do not display visual representation of a PDF document.

[4] `Acrobat Reader` is not supported on Linux. We used the last available version from 2013, which is newer than files in the data set.

[5] https://bugs.freedesktop.org/show_bug.cgi?id=97485

Table 4.3: Portfolio of PDF readers used in each phase.

Reader	Version	Phase 1	Phases 2	Phase3
pdfinfo	0.24.5	✓		
pdftk	2.01	✓		
ps2pdf	9.1	✓		
Xpdf	3.03-16	✓	✓	✓
Evince	3.10.3	✓	✓	✓
Okular	0.19.3	✓	✓	✓
qpdfview	0.4.7	✓	✓	✓
MuPDF	1.3-2	✓	✓	✓
Firefox	44.0.2		✓	
Chromium	48		✓	✓
Acrobat Reader	9.5.5		✓	✓

4.3.3 Infrastructure

Phase 1: Filtering. We load all documents by using readers from our portfolio. To speed up this time-consuming process, we leverage a cloud infrastructure. The infrastructure is a set of virtual machines running in an `Apache CloudStack`-based cloud. We used `Vagrant` with `VirtualBox` to create the machines. Each virtual machine is configured to have 16 CPUs running at 1GHz, with 16GB of RAM and a 80GB hard drive. We allocated around 30 such virtual machines in our cloud. Inside each machine there are 14 `LXC` containers, each of them representing a separate execution environment. Both the VMs and the containers are running `Ubuntu 14.04`. Each container is constrained to use only one core, leaving 2 free cores per each VM for the host OS. The containers are configured to run the graphical interface `LXDE` in headless mode using `Xdummy` or `Xvfb`.

We split the document set into a number of partitions. Each container is assigned one partition at a time. In our experiments we used a centralized storage mounted by all the VMs in the cloud via the `NFS` protocol.

We control the containers using `Ansible`, which is a cloud automation tool. Each container is loaded with an `Ansible` playbook (a sequence of commands) which performs the necessary setup and invokes a `Bash` script which executes the experiment.

To cluster error messages, we implemented a `Python` v3 script which extracts feature vectors from the captured standard error messages. The script makes use of the `scikit-learn`

package [217] for K-means clustering and the calculation of Silhouette coefficient.

We use `xdotool` to automatically perform the necessary graphical user interface interactions such as mouse clicks or getting the name of the window.

Phase 2 and 3: Inconsistency Detection and Localization. These two phases were performed on a different machine, equipped with an Intel i5-2400 3.10GHz CPU and 6GB of RAM, running `Ubuntu 14.04`. Due to technical problems, the screenshots for `Firefox` were captured in the cloud VMs. For related technical reasons, the `Firefox` screenshots were not used in inconsistency localization[6]. We use the `Xvfb` virtual screen buffer to open the files in high resolution.

To compare screenshots, we use `MATLAB` version `R2015` and two libraries: `Steerable Pyramids` [199] and `CW-SSIM` [44].

Performance. It takes around a week to load all the documents in the cloud. Once standard errors are captured, it takes several hours to cluster the files. Comparing two images with CW-SSIM in our setting takes 5s; we need seven comparisons (`Acrobat Reader` vs the rest) per file, which requires 35-40s per file. Inconsistency localization takes roughly 8-11 minutes per file. We believe these numbers are reasonable, give the data set size and that we operate on relatively large images (3000x3000 pixels before cropping).

4.3.4 Research questions.

In Sections 4.4, 4.5 and 4.6 we present a series of experiments. The experiments were conducted to answer the following research questions:

1. **Section 4.4**: How frequent are cross-reader inconsistencies and what are the main types of inconsistencies?
2. **Section 4.5**: How well does our automated technique perform in terms of finding cross-reader inconsistencies?
3. **Section 4.6**: How well does our inconsistency localization based on delta debugging perform?

[6] A bug in `Xvfb` prevented us to display the PDF files with Firefox in Fullscreen mode, making the screenshots incorrect.

4.4 A Study of Cross-reader Inconsistencies

We first manually study a reasonably large random sample of 2,313 PDF files from the `Govdocs1` database. This empirical study has two main goals. The first is to understand how severe cross-reader inconsistencies are, i.e., how often PDF files are displayed inconsistently and what the common symptoms are.

The second goal is to produce a random set of PDF files with ground-truth knowledge about cross-reader inconsistencies. Such information allows us to measure the standard metrics of precision and recall of the automated detection techniques: *precision* is the fraction of reported inconsistencies that are true inconsistencies, and *recall* or *true positive rate* is the fraction of all true inconsistencies that are detected. F1 is the harmonic mean of precision and recall.

4.4.1 Methods

It is impractical to manually examine the 230K files of the `Govdocs1` database. Thus, we study a one percent random sample of 2,313 documents. We open each document with the eight graphical PDF readers (described in §4.3.2) and identify any rendering differences. To reduce manual effort, we first automatically took screenshots of the displayed images in each reader. Then one of the authors manually compared the screenshots to determine if two screenshots generated from the same PDF file reveal any inconsistencies. If they do, we then manually open the file in the two readers and compare the displayed PDF images to confirm the inconsistency; it is to ensure that bugs in the screen capture tool do not affect our results.

On average, a PDF file in our sample has 28 pages, and more than 64,000 pages for all 2,313 files. For eight readers, there would be more than 500,000 images to check—prohibitively expensive for a manual analysis. Therefore, we focus on inconsistencies found on the first page of each document only.

Because we are not the authors of the PDF files, we do not know what is the correct rendering for each file. Therefore, we use the "majority" rule to classify inconsistent displays. If five readers or more have identical renderings and the others display the page differently, we consider the five readers to be correct and the others incorrect.

When a reader cannot render a non-embedded font, it will replace it with a default font. Most readers use different default fonts, and because there is no standardized behaviour on the correct rendering of a missing font, we cannot determine which reader behaves correctly. As a result, we only report the total number of such files.

Table 4.4: Number of consistent and inconsistent files. For the latter, we report how many readers behave similarly ("agree").

	# of files	%
Consistent with all readers	1999	86.5%
Inconsistent	314	13.5%
7/8 readers agree	128	41%
6/8 readers agree	92	29%
5/8 readers agree	38	12%
≤4/8 readers agree	56	18%

One may consider using `Acrobat Reader` as the ground-truth, as it is developed by the same company that created the PDF format. However, `Acrobat Reader` also has bugs, and furthermore the Linux version is no longer supported, with several known bugs present [3].

4.4.2 Inconsistencies are common

Table 4.4 shows that cross-reader inconsistencies are quite common—out of the 2,313 pages manually verified, 13.5% are rendered differently by at least one reader. Since these PDF files originate from US government websites, they are meant to be correctly read by users and were not generated to be displayed inconsistently in different readers on purpose. Yet, a significant portion is not. The bottom part of Table 4.4 shows the agreement between readers, e.g., in 41% of the cases 7 out of the 8 readers behave similarly.

To understand if our random sample size is big enough, we calculate the margin of error for this sample, which is 1% with 95% confidence. This means that with 95% confidence, the percentage of inconsistent files in the entire `Govdocs1` database would be 13.5% ±1%.

When projected on the entire 230K database, this percentage represents 28K to 33K inconsistent files, which can cause many of the miscommunication issues described in the introduction. Since only the first pages of those documents have been studied, and an inconsistency can happen anywhere in the document, the number of inconsistencies would likely be bigger than 13.5% when all pages of the documents are considered.

Table 4.5: Issues detected in the random sample, sorted by type and reader. Acrobat denotes `Acrobat Reader`.

Issue type	`Acrobat Reader`	`Chromium`	`Evince`	`Firefox`	`MuPDF`	`Okular`	`qpdfview`	`Xpdf`	**Total**
Performance issues	0	1	0	1	0	15	8	0	**25**
Missing images	0	1	0	1	26	0	0	0	**28**
Map bugs	0	0	2	4	0	0	0	0	**6**
Colour inconsistencies	44	19	5	28	7	3	0	0	**106**
Gradient inconsistencies	0	0	5	6	3	1	0	0	**15**
Form inconsistencies	12	12	0	0	0	0	0	0	**24**
Others	2	4	4	16	25	6	0	0	**57**
Font issues	178								
Total	58	37	16	56	61	25	8	0	**261**
Total Unique Bugs	3	5	4	9	8	2	0	0	**31**

4.4.3 Types of inconsistencies

Table 4.5 shows the different types of inconsistencies that we encountered. The numbers from this table do not map directly to the number of files triggering inconsistencies presented in Table 4.4, because one file can expose several issues in the same reader (e.g., missing an image and displaying an incorrect colour) or the same issue in multiple readers (considered as multiple bugs since they are present in multiple readers), and multiple files may expose the same issue in one reader (e.g., the 26 missing-image inconsistencies of `MuPDF` shown in Table 4.5 are all caused by a single `MuPDF` bug).

We describe each type of inconsistency below:

Performance issues. The first type of inconsistencies one notices when opening a PDF file concerns performance. This is not a graphical bug by itself, but our tool was able to detect it nonetheless. While we can ensure that the reader is fully loaded before taking the screenshot, it is not trivial to ensure the PDF page is fully rendered before taking a screenshot. If the reader takes too long to render the file, the screen capture might occur before the file is fully displayed. We decided to allow a reasonable time of 30s for the reader to display the file. If after this time the file is still not rendered correctly, we report a performance bug. Our results indicate that `MuPDF`, `Acrobat Reader`, `Evince` and `Xpdf` always render the PDF files within 30s, while the other PDF readers experience performance problems for some files.

Missing images. We observed several cases of missing images during our study. While in principle those issues could also be created by bugs in the PDF files (e.g., incorrectly compressed images), the 28 issues we encountered were caused by bugs in `Chromium`, `Firefox` and `MuPDF`. The bug in `MuPDF` was already known, while the other bugs were new and

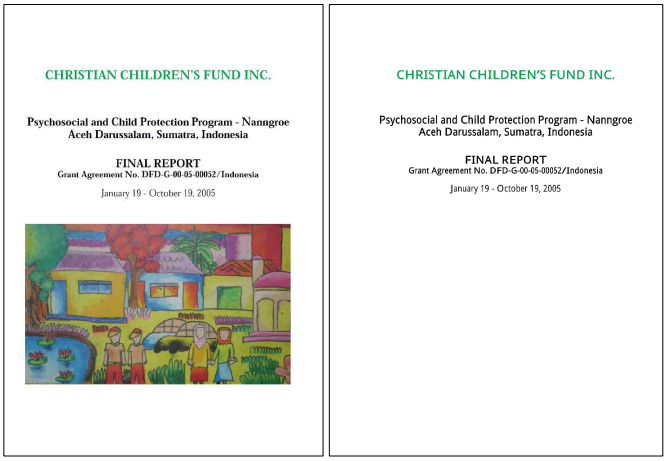


Figure 4.3: An example of a missing image bug in `MuPDF`. `MuPDF` on the right, other readers on the left.

confirmed by the developers. Figure 4.3 shows the missing image bug in `MuPDF`. This bug happened because of specific image encodings that were not well supported by the readers. This issue is critical when it occurs in scanned documents for which each page consists in one large image. In such cases, the buggy reader only displays blank pages.

Map bugs. Maps are challenging to render because they are generally made of several superposed layers of graphic vectors and often contain advanced features such as transparent objects. In the data set, we found several maps that were incorrectly rendered, similar to the example in Figure 4.1. These are bugs in the PDF readers.

Colour inconsistencies can either be bugs in the reader or in the PDF file. Figure 4.4 shows a colour bug in the Ubuntu version of `Acrobat Reader` 9.5.5. In this example, the background colour rendered by `Acrobat Reader` is different from the background colour rendered by other readers. Colour bugs can also be caused by buggy files, if the colour space of the file is incorrectly encoded. While no information is usually lost, this significantly affects the design of the document and can be a major issue for graphic designers.

Form inconsistencies. These inconsistencies occur when forms are included in the PDF file. `Chromium` and `Acrobat Reader` highlight editable fields while other readers do not. We believe such discrepancies are neither bugs in the PDF files, nor in the readers.

Others. This category contains a wide range of inconsistencies that appear rarely and are generally caused by bugs in a specific reader or rendering library. For example, we found a bug in `Evince` that occurs only in rare cases when a PDF file contains images that

Figure 4.4: Example of colour discrepancy between `Acrobat Reader` (left) and other readers (right)

have a colour depth inferior to 8 bits. Figure 4.5 displays an example of this bug. We filed this bug against the `Poppler` rendering library and the developers fixed it. [7]

Font inconsistencies. As we cannot know which readers display the correct font, we only report the total number of files that had at least one font inconsistency (178 in Table 4.5). Indeed, 178 out of 314 files that reveal inconsistencies contain a font inconsistency. This inconsistency occurs when an uncommon font is not embedded in the PDF file. In this case, the reader will either use a default font to replace the non-embedded font or simply not display the text. Figure 4.6 shows and example of this issue.

Reader reliability. Our experiments enable us to assess the relative reliability of the eight readers. The row *Total* of Table 4.5 shows the total number of bugs exposed in each reader, while the row *Total Unique Bugs* presents the number of unique bugs. We assessed uniqueness manually based on the type of inconsistency, potential identical warnings and error messages, the similarity of the documents, and whether the exact same set of readers behave similarly. For example, if two PDF files have the same inconsistency (e.g., a jpeg image is not displayed) with the same readers, then we consider that these two files reveal only one unique bug. On the other hand, if the image in the first file is incorrectly displayed with one specific reader (e.g., Evince) and the image in the second file is incorrectly displayed

[7] https://bugs.freedesktop.org/show_bug.cgi?id=94371

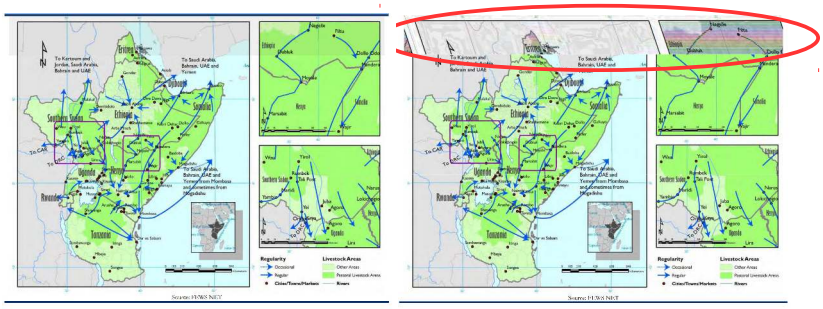


Figure 4.5: An example of “other" types of inconsistencies. `Chromium` rendering on the left, distorted `Evince` rendering on the right.

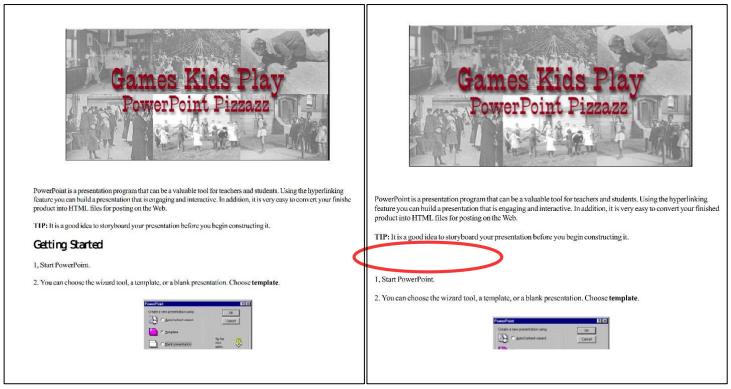


Figure 4.6: `Chromium` (left) can render the incorrectly embedded characters, while other readers (right) cannot.

with a different reader (e.g., Chromium), then these two PDF files reveal two unique bugs. Visual inconsistencies are often subjective, so best effort judgement seems to be a reasonable approach.

We can use both measures as an approximation of reader's reliability. With all other factors being equal, the higher the number of unique bugs, the lower the quality of a reader's codebase. On the other hand, under the assumption that the `Govdocs1` data set is representative of real-world documents (which we discuss in Section 4.3.1), the total number of bugs in each reader correlates with its frequency of failure in the field. Looking at these two metrics, `MuPDF` is the least reliable reader with 8 unique bugs and 61 bugs, while the most reliable reader is `Xpdf` with no bugs.

Summary. The experiments presented in this section try to answer the research question *How frequent are cross-reader inconsistencies and what are the main types of inconsistencies?* As we found out, cross-reader PDF inconsistencies are surprisingly common and we can categorize the inconsistencies into several common types. Finally, some PDF readers exhibit more visual inconsistencies than others.

4.5 Automatic results

In this section, we evaluate our automatic inconsistency detection technique described in Section 4.2. We devised two experiments, one using the random sample of documents that we manually inspected in Section 4.4, and the other using the entire 230K database.

4.5.1 Results for the random sample

We perform the first experiment on the sample of documents studied in Section 4.4[8]. This experiment evaluates the inconsistency detection phase of our approach: we perform image comparison on all files in the evaluation set without filtering to potentially identify all inconsistent PDF files. The aim of this experiment is to use the manually-determined ground truth to establish the precision and recall of our automated image comparison technique.

[8]For technical reasons we needed to exclude one file from the sample, because PDFBox was unable to parse it.

Our automatic tool detected 189 true inconsistent files, which revealed 21 unique bugs in the readers and 124 incorrect files. We consider a file as *incorrect* if it does not follow the PDF format specification or does not embed or subset non-standard fonts. While font embedding is not included in the PDF specifications, it is included in many publisher standards. For example, the minimum requirements for PDF published on IEEE Xplore platform include "embed or subset all fonts" [54].

The detection precision, recall and F1 for our approach are 33%, 60%, and 43% respectively. While 43% is the highest F1 value, we can tune the threshold of our technique to obtain different precision and recall values. In other words, because our inconsistency detection tool is based on a similarity score, we can modify the threshold to either reduce the number of false positives or false negatives. Figure 4.7 shows the ROC curve associated with the CW-SSIM similarity algorithm we used. This curve shows that we can get a reasonably good true positive rate (60%) while keeping a low false positive rate (20%). Increasing the true positive rate to 80% can be done if we accept to increase the false positive rate to 50%.

We also experimented with other image similarity algorithms: Absolute Error (AE), Mean Absolute Error (MAE), Mean Squared Error (MSE), square Root Mean Squared Error (RMSE), Normalized Cross Correlation (NCC), Peak Signal to Noise Ratio (PSNR) [89] and Perceptual Hash (PHASH) [270]. Figure 4.7 shows the ROC curves for each of them. CW-SSIM and PHASH are the best performing image comparison metrics. We select CW-SSIM as it was our first choice; it is also a bit faster (we did not perform a thorough time measurement but we believe that one second is a good estimate).

False positives: 73% of the false positive results are due to limitations of CW-SSIM. Some documents contain few or no structured elements (e.g., almost blank pages), making it difficult for CW-SSIM to identify structural similarities. These are edge cases where simpler techniques such as a histogram comparison might provide more accurate results. Future work could be done to use different image comparison algorithms depending on the visual features of the PDF file being evaluated. 11% of the false positives are due to GUI problems that are not correctly removed from the screenshots. In 10% of the cases, the files were correctly detected as inconsistent by our algorithm, but the manually generated ground truth was incorrect. The remaining false positive results were mostly due to spurious graphical bugs. We do not consider these spurious graphical bugs as true positives because their root cause is external to the PDF reader (e.g., bugs in the Unity graphical shell).

False negatives: Local inconsistencies such as font inconsistencies can be challenging to detect. For example, consider a document in which only a few words are rendered with an incorrect font; automatically detecting those few incorrectly rendered words is hard. This

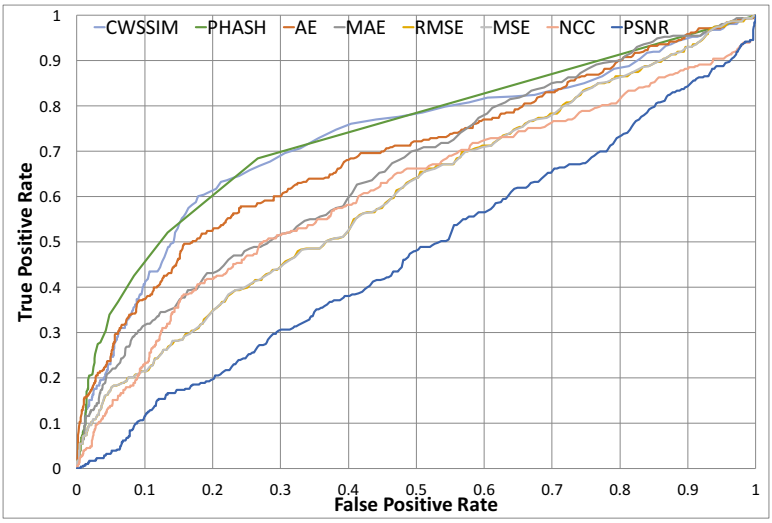


Figure 4.7: ROC curves for different image similarity algorithms.

concerns the vast majority of the inconsistent files classified as correct by our tool. Those "local" discrepancies are very hard to spot when looking at the entire pages because most of the page is actually rendered correctly. Possible improvements could be done by extracting the locations of the different text boxes from the PDF's metadata and running the image comparison on specific parts of the screenshot.

4.5.2 Results for the entire set

In the second experiment we evaluate our end-to-end approach, as described in Section 4.2. We analyze the whole data set of over 230,000 files: in the first stage, we load each document in every PDF reader used for this phase, detect crashes and capture emitted error messages. We then create clusters of documents for each reader based on the emitted error messages, randomly select one document from each cluster and run our inconsistency detection tool on the first page of the document. The reason for random selection of a single document from each cluster is the assumption that clustering should group documents with similar issues together. This allows to minimize the number of tested files to just one file per cluster; in our experiments, this reduces the test set size from 230K to 103. Although it is a heuristic, our experiments show that it yields good results.

Error messages. The number of documents in the `Govdocs1` data set that cause the readers to emit an error message of interest varies between 3,771 and 28,438, with an

Table 4.6: Number of crashes detected. The number of unique errors is shown in parentheses.

Reader	ps2pdf	pdftk	Evince	qpdfview	Others	Total
Crashes	2 (2)	5,281 (3)	112 (2)	8 (2)	0	5,403 (9)

Table 4.7: Inconsistencies for cluster candidates. The third column shows the number of inconsistent cluster candidates. The number of unique bugs is shown in parentheses.

Clusters for	**Number of clusters**	**Inconsistent candidates**	**Bugs triggered by candidates**
pdfinfo	5	2	2
pdftk	2	1	1
ps2pdf	19	7	7
Xpdf	9	2	2
Evince	41	16	23
Okular	2	2	3
qpdfview	13	4	5
MuPDF	12	6	6
Total	103	40	49 (10)

average of 15,293 and a median of 14,085 across the PDF readers. In total 65,406 files generated warnings or error messages in at least one of the readers, which accounts for 28% of documents in the set.

Crashes. In the first phase of our approach, we detect PDF reader crashes, such as segmentation faults and aborts. Table 4.6 presents the number of non-spurious crashes observed in ps2pdf, pdftk, Evince and qpdfview (the other readers had no crashes or only spurious ones).

Inconsistency bugs. Table 4.7 shows the number of clusters created for each PDF reader after running the filtering phase. This varies between only 2 clusters for pdftk and Okular and 41 for Evince, for a total of 103 clusters.

The column *Clusters for* indicates the PDF reader that generated the error messages that were used for clustering. The column *Number of clusters* presents the number of cluster obtained.

We randomly sampled one file (or candidate) from each cluster and compared the rendered images both manually and using our automated technique. We assume that the

file selected from the cluster is representative of the other files in the cluster. In the table we only display the results of manual inspection in order to illustrate the usefulness of clustering independently of the accuracy of image comparison, but we also present the results of our automatic tool below.

The column *Inconsistent candidates* reports the number of candidates that are inconsistent across the readers and the column *Bugs triggered by candidates* displays the number of bugs triggered by the inconsistent candidates. An inconsistent file can trigger different bugs in different readers, so this number might be higher than the number of inconsistent candidates.

For example, the first row shows that documents which generate error messages in `pdfinfo` were split into five clusters. From each of the clusters we randomly sample one candidate, resulting in a total of five candidates. Two out of these five candidates are inconsistent across PDF readers. Further manual examination of these two candidates indicates that they reveal two bugs.

In total, 40 out of 103 candidates have inconsistencies. Therefore 38% files selected via clustering are inconsistent compared to only 13.5% inconsistent files in our random sample discussed in Section 4.4. Across the inconsistent cluster candidates we found a total of 49 bugs, out of which 10 were unique. The rate of unique bugs to the number of files for cluster candidates is 10/ 103 = 9.7%, which is higher that the rate of 31/ 2,313 = 1.3% for the random sample. The higher percentages of inconsistent files and unique bugs suggest that the clustering approach is indeed effective.

Results of image comparison on the candidates: Finally, if we use the automatic image comparison-based inconsistency detection, we get 24 cluster representatives detected as inconsistent, which contain a total of 31 bugs, 6 unique bugs and 14 incorrect files.

Correlation with file producer: With `pdfinfo`, we retrieved the authorship data of the files producing standard errors or having non-zero return status. We extracted creator, producer and author of the file to see whether there is a correlation between inconsistent files and the software that produced them. The results are non-conclusive although there are many files that share common origins: there are ∼18,000 distinct creators, ∼1,400 distinct producers and ∼48,000 distinct authors.

We further analyzed the metadata of the files in the random sample, as presented in Table 4.4. First, we analyzed the correlation with the version of PDF document.

Table 4.8 presents the percentage of files for each PDF version in our sample, as well as the percentage of inconsistent files for each version. First, we can see the distribution of files across PDF versions in our random sample closely follows the distribution of our entire data set showed in Table 4.2.

Table 4.8: Comparison between the distribution of files across PDF versions in the entire random sample and in the inconsistent files from the random sample.

Percentage	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**
Whole random sample	0.6%	3.0%	16.8%	23.2%	37.8%	9.2%	8.8%	0.7%
Inconsistent files	2.5%	3.5%	20.4%	21.3%	25.8%	12.1%	13.7%	0.6%

Second, the distribution of inconsistent files across PDF versions roughly follows the distribution of the PDF versions in the sample, with a few exceptions. The largest difference occurs for PDF 1.4. Indeed, this version represents 37.8% of the random sample, but only 25.8% of the inconsistent files. Additionally, we measured the Pearson correlation between inconsistent files and PDF versions and did not find any correlation. For all PDF versions, the Pearson coefficient is between -0.1 and 0.1.

We also analyzed the correlation between creators and producers that were used to produce more than 20 documents in our random sample.

We found 12 creators used to generate more than 20 documents: InDesign, Pscript, Capture, PDFMaker, Print Server, PageMaker, POP90, Office, FrameMaker, WordPerfect, Microsoft Word, and QuarkXPress.

Only three creators have a small correlation with inconsistent PDF files. InDesign and QuarkXPress have a small positive correlation (Pearson coefficient between 0.1 and 0.3) with files rendered inconsistently. PScript presents a small negative correlation (Pearson coefficient between -0.1 and -0.3). For all the other creators, we found no correlation with incorrectly rendered documents.

There are seven producer fields used in more than 20 documents: Acrobat Distiller, Acrobat PDF Writer, Acrobat PDF Library, Ghostscript, Corel PDF Engine, Etymon, and PDFContext. None of them shows any significant correlation with inconsistently rendered PDF documents.

Summary of inconsistencies and bugs detected in both experiments: Our technique automatically detected inconsistencies and bugs in both the experiments in Section 4.5.1 and Section 4.5.2. In total, our approach detected 229 inconsistent files and 5,403 crashes/exceptions automatically, including 30 unique bugs in the readers and 138 incorrect files. The number of unique bugs (30) is not the sum of the number of bugs detected in each experiment, as some bugs overlap.

Bug reports: One of the outcomes of the project was filing 33 bug reports of which 17 were confirmed or fixed. We filed the bugs throughout the project, with some of them being

spotted during the development of the tool. We did not report all bugs, as some of them had already been reported or fixed.

The list of bug reports is presented on our project website at
https://srg.doc.ic.ac.uk/projects/pdf-errors/results.html.
Currently out of 33 reported bugs, 8 have already been fixed, 9 have been confirmed as true bugs, 5 have been triaged as "won't fix", and 11 await confirmation. Note that one of the fixed bugs (#17 on the list) was marked as a duplicate as there was a parallel report of a problem. We still count the bug towards the 33 unique bugs as that was our judgement at the time of reporting.

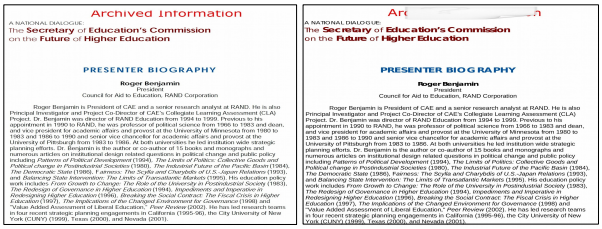
Summary. The experiments presented in this section try to answer the research question *How well does our automated technique perform in terms of finding cross-reader inconsistencies?* We show that the technique is able to find more inconsistencies in cluster candidates compared to a random selection of documents. We also show that our algorithm of choice for visual inconsistency detection is one of the two algorithms with the best ROC curve.

4.6 Inconsistency Localization Evaluation

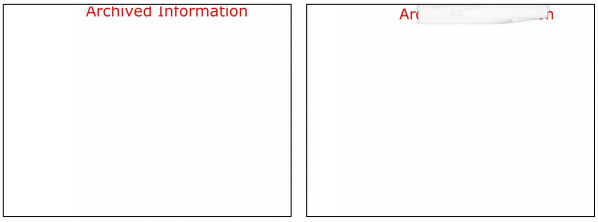
To evaluate the accuracy of our inconsistency localization technique, we run it on the 189 true inconsistent files detected by our approach on the random sample (see Section 4.5.1). The inconsistency localization algorithm was able to generate a reduced PDF file in 86% of cases. Most files (139) were reduced to only two visible objects, 13 to only one, and 21 to more than two.

We evaluated the correctness of the reduced PDF files by manually examining them to verify whether they actually reveal an inconsistency. 84% of the reduced PDF files display an element that is inconsistent. Figure 4.8a shows an example of inconsistent file incorrectly displayed by `Okular` and Figure 4.8b shows the same issue on the reduced file obtained after inconsistency localization. As we can see, the file has been reduced to only contain the inconsistent element.

43% of the incorrectly reduced PDF files are due to the inconsistency not being caused by a specific element. For example, `MuPDF` has a bug in its fullscreen functionality that does not work correctly when the size of the page is too large or too small compared to the screen's resolution, regardless of the elements in the file. The other 57% of the cases seem to be caused by false positives in CW-SSIM.



(a)



(b)

Figure 4.8: Inconsistency between Adobe Reader (left) and Okular (right) before (a) and after reduction (b).

In number of bytes, reduced files are 31% smaller than the original file. This is a relatively small reduction compared to the one in number of displayed objects, as a PDF file must contain many base structures to remain valid. We refer to the size of these base structures as the *base file size*. Using the base file size, we can also compute the relative size reduction, i.e., $\frac{reduced\ file\ size\ -\ base\ file\ size}{original\ file\ size\ -\ base\ file\ size}$. Our approach has a substantial relative size reduction of 86%.

Summary. The experiments presented in this section try to answer the research question *How well does our inconsistency localization based on delta debugging perform?* We show that in most cases the localization algorithm was able to reduce the "buggy" document to just a couple of visible objects.

4.7 Cross-OS Inconsistencies

Since there are more Windows users than Linux users, there is a possibility that developers of PDF readers spend more time improving the reliability of their PDF readers on Windows than on Linux, and it is possible that the inconsistencies we found are only a problem on Linux.

In this section, we aim at determining whether the files we detected as inconsistent on Linux readers also reveal bugs in Windows PDF readers. More specifically, we aim at answering the following two research questions:

Question 1: Are the inconsistencies we detected OS specific?

Question 2: Can we use the files we detected as inconsistent on Linux to detect bugs in another OS and other PDF readers?

4.7.1 PDF Readers on Windows

To understand whether the inconsistencies we found are OS specific, we chose, when possible, equivalent Windows versions of the software we tested on Linux. More precisely, `Firefox`, `Chromium` and `MuPDF` have an equivalent version for Windows. For `Xpdf` we only found a more recent version (4.00.1) while for `Evince`, only an older version was found (2.32.0.145). For `Acrobat Reader`, we chose to use the most recent version on Windows (Acrobat Reader DC) as it is likely to be the most used PDF reader on this OS. `Okular` and `qpdfview` do not have a Windows version available.

We completed this set of Windows PDF readers with two popular readers that are specific to Windows. The first one is `Edge`, Microsoft default browser on Windows 10, and the second one is `Sumatra PDF`, a popular lightweight PDF reader. In total we evaluated 8 PDF readers on Windows 10.

4.7.2 Experiment Details

For this experiment, we focused on the files we reported during our evaluation of Linux PDF readers. We reported 19 files revealing 22 different bugs. We focused on these files because these represent unique bugs that we found.

4.7.3 Results

Are the inconsistency we detected OS specific?

To answer this research question, we focus on bugs we reported for PDF readers that are cross-OS (e.g., `Firefox`, `Chromium`). The goal is to find out whether these bugs only occur in the Linux version or in both versions of the reader. When possible, we used the same

Table 4.9: Comparison between bugs on Linux and Windows version of PDF readers. # Linux column shows the number of bugs reported on Linux. # Windows represents the number and percentage of the reported bugs that also occur in the Windows version of the reader

PDF reader	# **Linux**	# **Windows**
Firefox	7	5 (71%)
Chromium	4	3 (75%)
mupdf	2	1 (50%)
Evince	3	3 (100%)
Total	16	12 (75%)

Table 4.10: Number of reported inconsistent files on Linux that also reveal bugs in Windows readers.

Acrobat	**Firefox**	**Chromium**	**Edge**	**Sumatra**	**Evince**	**mupdf**	**xpdf**
5/19	11/19	3/19	5/19	5/19	7/19	6/19	4/19

version of the reader (`Firefox`, `Chromium`, `MuPDF`). In the case of `Evince`, the Windows version is older than the Linux version.

The results displayed in Table 4.9 show that 75% of the bugs we reported for Linux also occur on Windows 10. The reason is that most of these bugs occur in the shared backend library that parses the PDF file and not in the OS specific source code.

Can we use the files we detected as inconsistent on Linux to detect bugs in another OS and other PDF readers?

To answer this question, we check how many of the inconsistent files we reported for Linux readers also reveal bugs on Windows readers. Table 4.10 displays the number of reported inconsistent files on Linux that also reveal bugs on Windows readers. For example, the Acrobat column indicates that 5 of the 19 inconsistent files on Linux readers reveal bugs in Acrobat Reader on Windows. We reported 11 bugs to the developers for the Windows readers (4 for `Acrobat Reader`, 3 for `Edge` and 4 for `Sumatra PDF`).

The files we reported for Linux helped us find inconsistencies in all the Windows readers tested. In particular, we found inconsistencies in Acrobat Reader DC, the most recent version of Acrobat Reader on Windows. We also found 5 inconsistencies in both Microsoft

Edge and Sumatra, two PDF readers that do not have an equivalent reader working on Linux.

This study highlights that the PDF inconsistency problem is not restricted to Linux readers, and that the files which are inconsistent on Linux readers are also likely to reveal inconsistencies in Windows readers.

4.8 Discussion and Threats to Validity

4.8.1 Conclusion Validity

Visual inconsistencies can be subjective. While most of them are clear (e.g., crashes and missing images), some are ambiguous (e.g., colour differences can be so small that some people consider them different while others see no difference). Therefore, there is a potential threat regarding the labelling of inconsistencies in our study. To mitigate this issue, the labelling was done by one of the authors, then independently verified by another author and two other students.

We empirically choose an optimal threshold and image comparison algorithm for our data set. However our study covers very diverse documents, and the optimal threshold or image comparison algorithm might be different for specific types of documents (e.g., image or text). In the future, we plan to take the content of the file into consideration for choosing the optimal threshold and image comparison algorithm to use.

4.8.2 Internal Validity

Despite its large size of about 230K files, the PDF files used in our evaluation come from a single source of the U.S. government's websites. Thus, the results on other PDF files may be different. However, the data set should be reasonably representative for real-world PDF files. To mitigate this issue, we did an extensive study on the data set, investigated the files' metadata and found out the files were produced by a wide range of software.

We mostly focus on PDF readers available on Linux. However, we claim the PDF inconsistency issue is generalizable to other platforms. To support this claim, we did a small study on the 18 files we reported for Linux PDF readers and found that these files also reveal inconsistencies in 8 different Windows PDF readers.

4.8.3 Construct Validity

If one PDF file exposes the same bug in all readers, and causes all readers to fail in the same way (e.g., all display an identical but wrong image), our approach would fail to detect this bug as there are no visual inconsistencies. However, in practice, we have not seen such cases during our manual examination.

4.8.4 External Validity

Although the presented technique is tuned for PDF documents, it is not bound to the PDF format. The same methodology of capturing error messages and return codes, clustering and then comparing screenshots could be applied to other document readers, like e.g., MS Word viewers, image viewers or web browsers. The aspects that need to be adjusted for a specific application, e.g., a different document type, are the regular expressions used for the emitted messages and the cropping functionality which removes application specific UI elements from the screenshots. The technique is applicable in a broader context because we treat programs in the portfolio in a black-box manner and capture their externally visible behaviour only.

More generally, the technique can be used to tackle the problems that arise when cross-checking a portfolio of programs on a large data set of inputs: which inputs to select (error messages), how to group similar inputs (clustering) and how to detect issues using a similarity metric (image comparison). The presented tool can be used by end-users to check that the published document is presented as expected and by developers to automatically detect bugs in a portfolio of viewers.

4.8.5 Practical Applicability

We envision the practical applicability of this research in two directions: 1) as empirical evidence/systematic study of the PDF inconsistency problem, and 2) as a technique/methodology for cross-testing a portfolio of reader programs and for cross-verification of document correctness.

Our empirical research highlights and quantifies the, otherwise anecdotal, problem of cross-reader PDF inconsistencies. The fact that these inconsistencies appear is an interesting problem on its own due to the intended portable nature of the PDF format. The results also show the difficulty of implementing a complex document format standard and a potential for inconsistencies to appear across different implementations of the standard. We hope

that our empirical study will spawn further research into document correctness and reader testing.

Our proposed inconsistency detection technique and prototype could be applied by:

- Developers to test their implementations of parsers and viewers. Our error clustering algorithm can help select interesting documents, the delta-debugging component can help to narrow down the problems and the minimized document can serve as a test case.
- Publishers/designers to make sure that the document looks consistently and as expected across multiple viewers (similarly to what is being done for web pages).
- End-users for sanity checks against bugs such as missing images or inability to load a file in certain readers.
- Other researchers who want to study similar inconsistency issues but find the data set is too large for thorough analysis.

4.9 Summary

This work presents and quantifies the research problem of cross-reader inconsistencies, which are caused by bugs in readers and files. We conduct an empirical study on 2,313 PDF files, which shows that cross-reader inconsistencies are common. In addition, we propose techniques to detect and localize inconsistencies automatically on over 230K PDF documents. Our approach has detected 30 unique bugs on Linux. We also reported 33 bugs to developers, 17 of which have already been confirmed or fixed.

Our database of clusters of PDF files and detected bugs, which we make available at http://srg.doc.ic.ac.uk/projects/pdf-errors, could help researchers and practitioners address software reliability challenges including testing other readers, and diagnosing and fixing bugs in readers and PDF files.

Chapter 5

Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques

5.1 Motivation

Software architecture is crucial for program comprehension, programmer communication, and software maintenance. Unfortunately, documented software architectures are either nonexistent or outdated for many software projects. While it is important for developers to document software architecture and keep it up-to-date, it is costly and difficult. Even medium-sized projects, of 70K to 280K source lines of code (SLOC), require an experienced recoverer to expend an average of 100 hours of work to create an accurate "ground-truth" architecture [65]. In addition, as software grows in size, it is often infeasible for developers to have complete knowledge of the entire system to build an accurate architecture.

Many techniques have been proposed to automatically or semi-automatically recover software architectures from software code bases [13, 42, 67, 111, 164, 240]. Such techniques typically leverage code dependencies to determine what implementation-level units (e.g., symbols, files, and modules) form a semantic unit in a software system's architecture. To understand their effectiveness, thorough comparisons of existing architecture recovery techniques are needed. Among the studies conducted to evaluate different architecture recovery techniques [13, 165, 256], the latest study [64] compared nine variants of six existing architecture recovery techniques. This study found that, while the accuracy of the recovered architectures varies and some techniques outperform others, their overall accuracy is low.

This previous study used *include dependencies* as inputs to the recovery techniques. These are file-level dependencies established when one file declares that it includes another file. In general, the include dependencies are inaccurate. For example, file `foo.c` may declare that it includes `bar.h`, but may not use any functions or variables declared or defined in `bar.h`. Using include dependencies, one would conclude that `foo.c` depends on `bar.h`, while `foo.c` has no actual code dependency on `bar.h`.

In contrast, *symbol dependencies* are more accurate. A symbol can be a function or a variable name. For example, consider two files `Alpha.c` and `Beta.c`: file `Alpha.c` contains method `A`; and file `Beta.c` contains method `B`. If method `A` invokes method `B`, then method `A` depends on method `B`. Based on this information, we can conclude that file `Alpha.c` depends on file `Beta.c`.

A natural question to ask is, to what extent would the use of symbol dependencies affect the accuracy of architecture recovery techniques? We aim to answer this question empirically, by analyzing a set of real-world systems implemented in Java, C, and C++.

Dependencies can be grouped to different levels of granularity, which can affect the manner in which recovery techniques operate. Generally, dependencies are extracted at the file level. For large projects, dependencies can be grouped to the module level, where a module is a semantic unit defined by system build files. Module dependencies can be used to recover architectures even when finer-grained dependencies do not scale. In this work, we study the extent to which the granularity of dependencies affects the accuracy of architecture recovery techniques.

Another key factor affecting the accuracy of a recovery technique is whether dependencies utilized as input to a technique are direct or transitive. Transitive dependencies can be obtained from direct dependencies by using a transitive-closure algorithm, and may add relationships between strongly related components, making it easier for recovery techniques to extract such components from the architecture. However, as the number of dependencies increases, the use of transitive dependencies with some recovery techniques may not scale to large projects.

Different symbols can be used (functions, global variables, etc.) to create a symbol dependency graph, but it is unclear which symbols have the most impact on the accuracy of architecture recovery techniques. In this work, we study the impact of function calls and global variable usage on the quality of architecture recovery techniques. In addition, both C++ and Java offer the possibility of using dynamic-bindings mechanisms. Several techniques exist to build dynamic-bindings graphs [18, 46, 227] and, despite the existence of two early studies [84, 203] about the impact of call-graph construction algorithms, and the origins of software dependencies on basic architecture recovery, no work has been done to

study the effect of dynamic-bindings resolution on recent architecture recovery techniques.

The last question we study pertains to the scalability of existing automatic architecture recovery techniques. While large systems have been studied and their architectures analysed in previous work [24, 55, 236], the largest software system used in the published evaluations of automatic architecture recovery techniques is Mozilla 1.3, comprising 4MSLOC, and it revealed the scalability limits of several recovery techniques [64]—an old version of Linux was also studied, but its size reported in previous evaluations was only 750KSLOC. The size of software is increasing, and many software projects are significantly larger than 4MSLOC. For example, the Chromium open-source browser contains nearly 10MSLOC. In this work, we test whether existing automatic architecture recovery techniques can scale to software of such size.

To this end, this work compares the *same* nine variants of six architecture recovery techniques from the previous study [64], as well as two additional baseline algorithms, using eight different types of dependencies on five software projects to answer the following research questions (RQ):

RQ1: Can more accurate dependencies improve the accuracy of existing architecture recovery techniques?

RQ2: What is the impact of different input factors, such as the granularity level, the use of transitive dependencies, the use of different symbol dependencies and dynamic-bindings graph construction algorithms, on existing architecture recovery techniques?

RQ3: Can existing automatic architecture recovery techniques scale to large projects comprising 10MSLOC or more?

This work makes the following contributions:

- We compared nine variants of six architecture recovery techniques using eight types of dependencies at different levels of granularity to assess their effects on accuracy. More specifically, we studied the impact of dynamic-bindings resolution algorithms, function calls, and global variable usage on the accuracy of recovery algorithms. We also expand the previous work by studying whether using a higher level of granularity or transitive dependencies improve the accuracy of recovery techniques. This is the first substantial study to the impact of different types of dependencies for architecture recovery.

- We found that the types of dependencies and the recovery algorithms have a significant effect on recovery accuracy. In general, symbol dependencies produce software architectures with higher accuracy than include dependencies (**RQ1**). Our results suggest

that, apart from the selection of the "right" architecture recovery techniques, other factors to consider for improved recovery accuracy are the dynamic-bindings graph resolution algorithm, the granularity of dependencies, and whether such dependencies are direct or transitive (**RQ2**).

- Our results show that the accuracy is low for all studied techniques, with only one technique (ACDC) consistently producing better results than k-means, a basic machine learning algorithm. This corroborates past results [64] but does so on a different set of subject systems, including one significantly larger system, and for a different set of dependency relationships.

- We recovered the ground-truth architecture of Chromium (svn revision 171054). This ground-truth architecture was not available previously and we obtained it through two years of regular discussions and meetings with Chromium developers. We also updated the architectures of Bash and ArchStudio that were reported in [65]. All ground-truth architectures have been certified by the developers of the different projects.

- We propose a new submodule-based architecture recovery technique that combines directory layout and build configurations. The proposed technique was effective in assisting in the recovery of ground-truth architectures. Compared to FOCUS [49], which is used in previous work [65], to recover ground-truth architectures, the submodule-based technique is conceptually simple. Since the technique is used for generating a starting point, its simplicity can be beneficial; any issues potentially introduced by the technique itself can later be mitigated by the manual verification step.

- We found some recovery techniques do, and some do not, scale to the size of Chromium. Working with coarser-grained dependencies and using direct dependencies are two possible solutions to make those techniques scale (**RQ2 and RQ3**).

5.2 Approach

Our approach is illustrated in Figure 5.1. First, we extract different types of dependencies for each projects. Then, we provide those dependencies as input to six different architecture recovery techniques. We also evaluate three additional techniques that take the project's source code as input. Finally, we used K-means results and architectures extracted from the directory structure of the project as a baseline. To evaluate the quality of the architecture recovered from different sets of dependencies, we obtain a ground-truth architecture of each project that was certified by each project's developers or main architect. Then, we

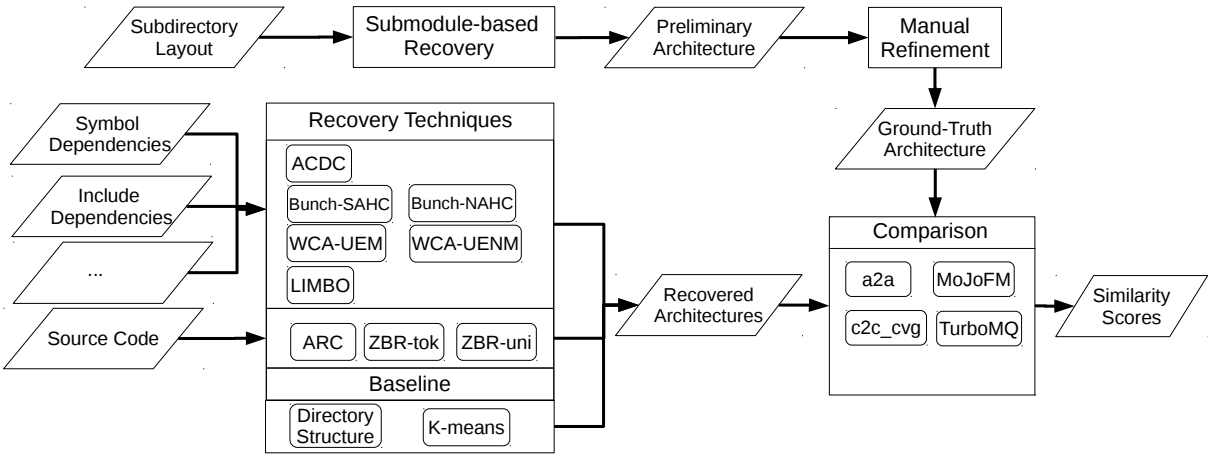


Figure 5.1: Overview of our approach

measure the quality of the architectures recovered automatically by comparing them to the ground-truth architecture using four different metrics.

In the rest of this section, we describe the manner in which we extract the dependencies we study, and elaborate on our approach for obtaining ground-truth architectures.

5.2.1 Obtaining Dependencies

Both symbol dependencies and include dependencies represent relationships between files, but the means by which these dependencies are determined vary.

C/C++ Projects

To extract symbol dependencies for C/C++, we use the technique built by our team that scales to software systems comprising millions of lines of code [247]. The technique compiles a project's source files into LLVM bitcode, analyzes the bitcode to extract the symbol dependencies for all symbols inside the project, and groups dependencies based on the files containing the symbols. At this stage, our extraction process has not considered symbol declarations. As a result, header-file dependencies are often missed because many header files only contain symbol declarations. To ensure we do not miss such dependencies, we augment symbol dependencies by analyzing #include statements in the source code.

These symbol dependencies are direct dependencies, which may be used at the file level or grouped at the module level. For large projects such as Chromium [1] and ITK [2], many developer teams work independently on different parts of the project. To facilitate this work, developers divided these projects into separated sections (modules) that can be updated independently. To group code-level entities at the module level, we extract module information from the build files of the project provided by the developers (e.g. makefile or equivalent). Transitive dependencies are obtained for all projects using the Floyd-Warshall [61] algorithm on symbol dependencies. Because the Floyd-Warshall algorithm did not scale for Chromium, we also tried to use Crocopat [21] to obtain transitive dependencies for Chromium and encountered similar scalability issues.

To extract include dependencies we use the compiler flag `-MM`. Include dependencies are similar to the dependencies used in prior work [64].

Java Projects

To extract symbol dependencies for Java, we leverage a tool that operates at the Java bytecode level and extracts high-level information from the bytecode in a structured and human readable format [202]. This allows for method calls and member access (i.e., relationships between symbols) to be recorded without having to analyze the source code itself. Using this information provides a complete picture of all used and unused parts of classes to be identified. We can identify which file any symbol belongs to, since the Java compiler follows a specific naming convention for inner classes and anonymous classes. With information about usage among symbols and resolving the file location for each symbol, we can build a complete graph of the symbol dependencies for the Java projects. This method accounts only for symbols used in the bytecode and does not account for runtime usage which can vary due to reflective access.

We approximate include dependencies for Java by extracting import statements in Java source code by utilizing a script to determine imports and their associated files. To ensure we capture potential build artifacts, the Java projects are compiled before extracting import statements. The script used to extract the dependencies detects all the files in a package. Then for every file, it evaluates each import statement and adds the files mentioned in the import as a dependency. When a wildcard import is evaluated, all classes in the referred package are added as dependencies.

[1] https://www.chromium.org/developers/how-tos/chromium-modularization
[2] https://itk.org/Wiki/ITK/Release_4/Modularization

The Java projects studied do not contain well-defined modules. In addition, our ground-truth architecture is finer-grained than the package level. For example, Hadoop ground-truth architecture contains 67 clusters when the part of the project we study contains only 52 packages. Therefore, we cannot use Java packages as an equivalent of C++ modules for our module-level evaluation for those specific projects. When studying larger Java projects, using Java packages could be a good alternative to modules defined in the configuration files used for C++ projects.

Relative Accuracy of Include and Symbol Dependencies

C/C++ include dependencies tend to miss or over-approximate relationships between files, rendering such dependencies inaccurate. Specifically, include dependencies over-approximate relationships in cases where a header file is included but none of the functions or variables defined in the header file are used (recall Section 5.1).

In addition, include dependencies ignore relationships between non-header files (e.g., .cpp to .cpp files), resulting in a significant number of missed dependencies. For example, consider the case where `A.c` depends on a symbol defined in `B.c` because `A.c` invokes a method defined in `B.c`. Include dependencies will not contain a dependency from `A.c` to `B.c` because `A.c` includes `B.h` but not `B.c`. For example, in Bash, we only identified 4 include dependencies between two non-header files, although there are 1035 actual dependencies between non-header files based on our symbol results. Include dependencies miss many important dependencies since non-header files are the main semantic components of a project.

A recovery technique can treat non-header and header files whose names before their extensions match (e.g., `B.c` and `B.h`) as a single unit to alleviate this problem. However, this remedy does not handle cases where such naming conventions are not followed or when the declarations for types are not in a header file.

Include dependencies use transitive dependencies for header files. Consider an example of three files `A.c`, `A.h`, and `B.h`, where `A.c` includes `A.h` and `A.h` includes `B.h`; `A.c` has an include dependency with `B.h` because including `A.h` implicitly includes everything that `A.h` includes.

For Java projects, include dependencies miss relationships between files because they do not account for intra-package dependencies or fully-qualified name usage. At the same time, include dependencies can represent spurious relationships because some imports are unused and wildcard imports are overly inclusive. Include dependencies are therefore significantly less accurate than symbol dependencies.

Overall Accuracy of Symbol Dependencies

To ensure the symbol dependencies we extracted are accurate, we randomly sampled 0.05% of the symbol dependencies and investigated whether these dependencies are correct. This small sample represents 343 dependencies we manually verified. The sampling was done uniformly across projects and dynamic bindings resolutions (interface-only or class hierarchy analysis). We did not find any incorrectly extracted dependencies in this sample, the margin of error being 5.3% with 95% confidence.

We did not quantitatively check whether all the existing dependencies were extracted, as it would be extremely time-consuming to do. However, when building the tool used for extracting dependencies [246], qualitative sanity checks were done to make sure the tool did not miss obvious dependencies.

5.2.2 Obtaining Ground-Truth Architectures

To measure the accuracy of existing software architecture recovery techniques, we need to know a "ground-truth" architecture of a target project. Since it is prohibitively expensive to build architectures manually for large and complex software, such as Chromium, we use a semi-automated approach for ground-truth architecture recovery.

We initially showed the architecture recovered using ACDC to a Chromium developer. He explained that most of the ACDC clusters did not make sense and suggested that we start by considering module organization in order to recover the ground truth.

In response, we have introduced a *simple submodule-based approach* to extract automatically a preliminary ground-truth architecture by combining directory layout and build configurations. Starting from this architecture, we worked with developers of the target project to identify and fix mistakes in order to create a ground-truth architecture.

The submodule-based approach groups closely related modules, and considers which modules are contained within another module. It consists of three steps. First, we determine the module that each file belongs to by analyzing the configuration files of the project.

Second, we determine the submodule relationship between modules. We define a *submodule* as a module that has all of its files contained within the subdirectory of another module. We first determine a module's *location*, which is defined as the common parent directories that contain at least one file belonging to the module. Then we can determine if a particular module has a relation to another module.

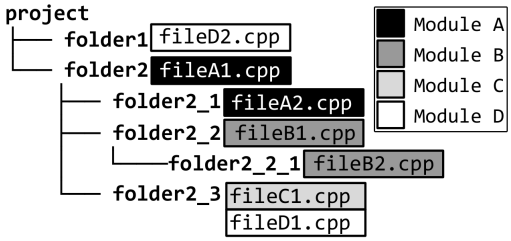


Figure 5.2: Example Project Layout

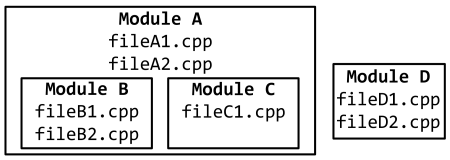


Figure 5.3: Example Project Submodules

For example, assume a project has four modules named A, B, C, and D. The file structure of the project is shown in Figure 5.2, while the module structure that we generate is shown in Figure 5.3.

- **Module A:** contains `fileA1.cpp` and `fileA2.cpp`. Location is `project/folder2`.
- **Module B:** contains `fileB1.cpp` and `fileB2.cpp`. Location is `project/folder2/folder2_2`.
- **Module C:** contains `fileC1.cpp`. Location is `project/folder2/folder2_3`.
- **Module D:** contains `fileD1.cpp` and `fileD2.cpp`. Location is both `project/folder1` and `project/folder2/folder2_3`.

Based on the modules' locations, we determine that module B is a submodule of module A because module B's location `project/folder2/folder2_2` is within module A's location `project/folder2`. Similarly, module C is a submodule of module A. The reason module D has two folder locations is because there is no common parent between the two directories. If module D had a file in the project folder, then its location would simply be `project`. Module D is not a submodule of module A because it has a file located in `project/folder1`.

This preliminary version of the ground-truth architecture does not accurately reflect the "real" architecture of the project and additional manual corrections are necessary. For example, Chromium has two modules `webkit_gpu`, located in the folder `webkit/gpu`, and

`content_gpu`, located in the folder `content/gpu`. The two modules are in completely separate folders and are grouped in different clusters by the submodule approach. However, both are involved with displaying GPU-accelerated content and should be grouped together to indicate their close relationship to the `gpu` modules. This is an example where the submodule approach based on folder structure may not accurately reflect the semantic structure of modules and needs to be manually corrected.

Hundreds of hours of manual work are then required to investigate the source code of the system to verify and fix the relationships obtained. When we are satisfied with our ground-truth version, we send to the developers the list of clusters containing files and modules in the Rigi Standard Format and a visual representation of how the clusters interact with one another for certification. Multiples rounds of verifications, based on developers' feedback, are necessary to obtain an accurate ground-truth architecture. For the recovery of Chromium, we also had several in-person meetings with a Chromium developer where he explained to us his view of the project's architecture and updated the parts of our preliminary architectures that were inaccurate. During these meetings, the Chromium developer investigated those clusters to see if they make sense (for example, whether the cluster names match with his understanding of Chromium modules and clusters). Then we showed him which files belongs to each cluster using different visualizations (e.g. the "spring" model from Graphviz [63] and a circular view using d3 Javascript library [234]), and he also verified if they were correctly grouped. When he did not agree, we checked if there was some mistakes on our side (i.e. inaccuracy in the submodule technique) or if it was a bug in the Chromium module definition.

It took *two years* of meetings and email exchanges with Chromium developers to obtain a ground truth.

The final ground truth we obtain is a nested architecture. Because most of the architecture recovery techniques produce a flat architecture, we flatten our ground-truth architecture by grouping modules that are submodules of one another into a cluster. In the example above, we cluster modules A, B and C into a single cluster and leave module D on its own.

Previous work [49, 65] mentioned there might exist different ground-truth architectures for the same project. Despite the fact that our submodule-based approach only recover one ground truth, it is possible to use our approach as a starting point for recovering several ground truths, by having different recoverers and receiving feedback from different developers.

Prior work [65] used a different approach, FOCUS [49], to recover preliminary versions of ground-truth architectures. Compared to FOCUS, the proposed submodule-based technique

is conceptually simpler. However, the submodule-based technique uses the same general strategy as FOCUS and can, in fact, be used as one of FOCUS's pluggable elements. This fact, along with the extensive manual verification step, suggests that the strategy used as the starting point for ground-truth recovery does not impact the resulting architecture (as already observed in [65]).

5.3 Selected Recovery Techniques

We select the same nine variants of six architecture recovery techniques as in previous work [64] for our evaluation. We also used 2 baseline clustering algorithms, the K-means algorithm and a directory-based recovery technique. Four of the selected techniques (ACDC, LIMBO, WCA, and Bunch [164]) use dependencies to determine clusters, while the remaining two techniques (ARC and ZBR [42]) use textual information from source code. We include techniques that do not use dependencies to (1) assess the accuracy of finer-grained, accurate dependencies against these information retrieval-based techniques and to (2) determine their scalability.

The view that the techniques we evaluate recover are structural views representing components and their configurations. Such views are fundamental and should be as correct as possible before making other architectural decisions. For example, behavioral or deployment views are still highly dependent on accurate component identification and the configurations among components.

Algorithm for Comprehension-Driven Clustering (ACDC) [240] is a clustering technique for architecture recovery. We included ACDC because it performed well in several previous studies [13, 64, 165, 256]. ACDC aims to achieve three goals. First, to help understand the recovered architecture, the clusters produced should have meaningful names. Second, clusters should not contain an excessive number of entities. Third, the grouping is based on identified patterns that are used when a developer describes the components of a software system. The main pattern used by ACDC is called the "subgraph dominator pattern". To identify this pattern, ACDC detects a dominator node n_0 and a set of nodes $N = \{n_i \mid i \in \mathbb{N}\}$ that n_0 dominates. A dominator node n_0 dominates another node n_i if any path leading to n_i passes through n_0. Together, n_0, N, and their corresponding dependencies form a subgraph. ACDC groups the nodes of such a subgraph together into a cluster.

Bunch [163, 164] is a technique that transforms the architecture recovery problem into an optimization problem. An optimization function called Modularization Quality

Table 5.1: Evaluated projects and architectures. †Cluster denotes the number of clusters in the ground-truth architectures. N/A means the value is not available.

Project	Version	Description	SLOC	File	Cluster†	Inc Dep.	Sym Dep.	Trans Dep.	Mod Dep.
Chromium	svn-171054	Web Browser	10M	18,698	67	1,183,799	297,530	N/A	4,455
ITK	4.5.2	Image Segmentation Toolkit	1M	7,310	11	169,017	30,784	19,281,510	2,700
Bash	4.2	Unix Shell	115K	373	14	2,512	2,481	26,225	N/A
Hadoop	0.19.0	Data Processing	87K	591	67	1,656	3,101	79,631	N/A
ArchStudio	4	Architecture Development	55K	604	57	866	1,697	10,095	N/A

(MQ) represents the quality of a recovered architecture. Bunch uses hill-climbing and genetic algorithms to find a partition (i.e., a grouping of software entities into clusters) that maximizes MQ. As in previous work [64], we evaluate two versions of the Bunch hill-climbing algorithms—Nearest and Steepest Ascent Hill Climbing (NAHC and SAHC).

Weighted Combined Algorithm (WCA) [166] is a hierarchical clustering algorithm that measures the inter-cluster distance between software entities and merges them into clusters based on this distance. The algorithm starts with each entity in its own cluster associated with a feature vector. The inter-cluster distance between all clusters is then calculated, and the two most similar clusters are merged. Finally, the feature vector of the new cluster is recalculated. These steps are repeated until WCA reaches the specific number of clusters defined by the user. Two measures are proposed to measure the inter-cluster distance: Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM). The main difference between these measures is that UENM integrates more information into the measure and thus might obtain better results. In our recent study [64], UE and UENM performed differently depending on the systems tested, therefore, we evaluate both.

LIMBO [12] is a hierarchical clustering algorithm that aims to make the Information Bottleneck algorithm scalable for large data sets. The algorithm works in three phases. Clusters of artefacts are summarized in a Distributational Cluster Feature (DCF) tree. Then, the DCF tree leaves are merged using the Information Bottleneck algorithm to produce a specified number of clusters. Finally, the original artefacts are associated with a cluster. The accuracy of this algorithm was evaluated in several studies. It performed well in most of the experiments [13, 165], except in one recent study [64] where LIMBO achieved surprisingly poor results.

Architecture Recovery using Concerns (ARC) [67] is a hierarchical clustering algorithm that relies on information retrieval and machine learning to perform a recovery. This technique does not use dependencies and is therefore not used to evaluate the influence of different levels of dependencies. ARC considers a program as a set of textual documents and utilizes a statistical language model, Latent Dirichlet Allocation (LDA) [22], to extract concerns from identifiers and comments of the source code. A concern is as a role, concept

or purpose of the system studied. The extracted concerns are used to automatically identify clusters and dependencies. ARC is one of the two best-scoring techniques in our previous evaluation [64] and thus is important to compare against when evaluating for accuracy.

Similar to ARC, **Zone Based Recovery (ZBR)** [42] is a recovery technique based on natural language semantics of identifiers and comments found in the source code. Each file is represented as a textual document and divided into zones. For each word in a zone, ZBR evaluates the term frequency-inverse document frequency (tf-idf) score. Each zone is weighted using the Expectation-Maximization algorithm. ZBR has multiple methods for weighting zones. The initial weights for each zone can be uniform (ZBR-uni), or set to the ratio of the number of tokens in the zone to the number of tokens in the entire system (ZBR-tok). We chose these two weighting variations to ensure consistency with the previous study [64]. The last step of ZBR consists of clustering this representation of files by using group-average agglomerative clustering. ZBR demonstrated accuracy in recovering Java package structure [42] but struggled with memory issues when dealing with larger systems [64].

Previous techniques are clustering algorithms specifically designed for architecture recovery. To obtain an estimate of the quality of the architectures generated by these algorithms, we used two baselines. For the first baseline, we cluster the files using the **K-means** algorithm. Each entity (i.e., a file or module) is represented by a feature vector $\{f_1,f_2,...,f_n\}$, where n is the number of features. Each of the n features represents a dependency with one of the n entities in the project.

For the second baseline, we used the **directory structure of the project** as an approximation of the architecture of the software. If automatic architecture recovery techniques cannot generate a recovered architecture that is superior to the directory structure of the project, then the recovery technique is not helpful for the specific project. To generate this approximated architecture, we use the same implementation as previous work [123].

5.4 Experimental Setup

In this section, we describe our experimental environment, how we obtained the ground-truth architectures for each project, the parameters used in our experiments, and the different metrics used to assess the quality of the recovered architectures.

5.4.1 Projects and Experimental Environment

We conduct our comparative study on five open source projects: Bash, ITK, Chromium, ArchStudio, and Hadoop. Detailed information about these projects can be found in Table 5.1. We choose those specific version of Bash and Chromium because they were the most recent versions available when we started our ground-truth recovery. For ArchStudio, Hadoop and ITK, we picked those versions because their respective ground-truth architectures were already available.

To run our experiments, we leveraged two machines and parallel processing, due to the large size of some projects. We ran ZBR with the two weight variations described in Section 5.3 on a 3.2GHz i7-3930K desktop with 12 logical cores, 6 physical cores, and 48GB of memory. We ran all the other recovery techniques on a 3.3GHz E5-1660 server with 12 logical cores, 6 physical cores, and 32GB memory.

For Bash, Hadoop, and ArchStudio, all techniques take a few seconds to a few minutes to run. For large projects, such as ITK and Chromium, each technique takes several hours to days to run. Running all experiments for Chromium would take more than 20 days of CPU time on a single machine. Consequently, we parallelized our experiments.

5.4.2 Extracted Dependencies

For the C/C++ projects, the number of include dependencies is much larger than the number of symbol dependencies, e.g., 297,530 symbol dependencies versus 1,183,799 include dependencies for Chromium. This is the result of both transitive and over-approximation of dependencies, detailed in Section 5.2.1.

The number of transitive dependencies shown in Table 5.1 for ITK is strikingly high. We leverage Class Hierarchy Analysis (CHA) [46] to build the dynamic-bindings dependency graph of symbol dependencies which, in turn, is used for extracting dependencies. When using CHA, we consider that each time a method from a specific class is called, all its subclasses are also called. Depending on how the developers use dynamic bindings, this can generate a large number of dependencies. For example, for ITK, more than 75% of the dependencies extracted are virtual function calls, as opposed to just 11% for Chromium. This high proportion of dynamic bindings also results in an extremely large number of transitive dependencies.

For Chromium, the algorithm to obtain transitive dependencies ran out of memory on our 32GB server. None of the recovery technique scaled for ITK with transitive dependencies. Given that Chromium is around ten times larger than ITK, it is safe to assume that, even

if we were able to obtain the transitive dependencies for Chromium, none of the technique would scale to Chromium with transitive dependencies.

5.4.3 Ground-truth architectures

To assess the effect of different types of dependencies on recovery techniques, we obtained ground-truth architectures for each selected project. Compared to previous work [64], we do not use Linux 2.0.27 and Mozilla 1.3 because our tool that extracts symbol-level dependencies for C++ projects works with LLVM. Making those two projects compatible with LLVM would require heavy manual work. In place of those medium-sized projects, we included ITK. We also included a very large project, Chromium, for which we recovered the ground truth. Due to issues resolving library dependencies with an older version of OODT, for which a ground-truth architecture is available [65], we were unable to use it for our study.

For Chromium, the ground-truth architecture was obtained by manually improving the preliminary architecture extracted using the submodule approach outlined in Section 5.2.2. After several updates and meetings with a Chromium developer, the ground-truth architecture was certified by one of Chromium's main developers. ITK was refactored in 2013 and its ground-truth architecture, extracted by ITK's developers, is available. We contacted one of ITK's lead developers involved in the refactoring who confirmed that this architecture was still correct for ITK 4.5.2.

The version of Bash used in a recent architecture-recovery study [64] was from 1995. Bash has been changed significantly since then (e.g., from 70KSLOC to 115KSLOC). Therefore, we recovered the ground-truth architecture of the latest version of Bash and used it in our study. Our certifier for Bash is one of Bash's primary developers and its sole maintainer, who also recently authored a chapter on Bash's idealized architecture [25].

The ground-truth architecture for ArchStudio was updated, from prior work [65], to be defined at the file level instead of at the class level. Additionally, ArchStudio's original ground-truth architecture had a number of inconsistencies and missing files, which were verified and corrected by ArchStudio's primary architect.

Hadoop, an open-source Java project used in a recent architecture-recovery study [64], was the other Java project we evaluated. Its original ground-truth architecture was based on version 0.19.0 and had to be converted from the class level to the file level for our analysis. For our analysis, we focused on the HDFS, Map-Reduce, and core parts of Hadoop.

5.4.4 Architecture Recovery Software and Parameters

To answer the research questions, we compare the clustering results obtained from nine variants of the six architecture recovery techniques, using include and symbol dependenciesdifferent types of dependencies. All input dependencies and output recovered architectures are generated in the Rigi Standard Format [226]. We obtained ACDC and Bunch from their authors' websites. The K-means-based architecture recovery technique was implemented using the scikit-learn python library [195]. For the other techniques, we used our implementation from our previous study [64, 123]. Each of those implementations was shared with the original authors of the recovery techniques and confirmed as correct [64]. Due to the non-determinism of the clustering algorithms used by ACDC and Bunch, we ran each algorithm five times and reported the average results. WCA, LIMBO, ARC and K-means can take varying numbers of clusters as input. We experimented with 20 clusters bellow and above the number of clusters in the ground truth, with an increment of 5 for all cases. For example, for ArchStudio, we ran these algorithms for 40 to 80 clusters. ARC also takes a varying number of concerns as input. We experimented with 10 to 150 concerns in increments of 10. We report the average results for each technique.

5.4.5 Accuracy Measures

There might be multiple ground-truth architectures for a system [23, 65]; that is, experts might disagree. Therefore, a recovered architecture may be different from a ground-truth architecture used in this work, but close to another ground-truth architecture of the same project. To mitigate this threat, we selected four different metrics commonly used in other automatic architecture recovery evaluations to measure the impact of the different inputs on the quality of the recovered architectures.

One of the metrics—normalized TurboMQ—is independent of any ground-truth architecture, which calculates the quality of the recovered architectures. When we use normalized TurboMQ to compare different recovery techniques, the threat of multiple ground-truth architectures should not apply. The remaining three metrics—MoJoFM, a2a and $c2c_{cvg}$—calculate the similarity between a recovered architecture and a ground-truth architecture. If one recovery technique consistently performs well according to all metrics, it is less likely due to the bias of one metric or the particular ground-truth architecture. Although using four metrics cannot eliminate the threat of multiple ground-truth architectures entirely, it should give our results more credibility than using MoJoFM alone.

We used MoJoFM, a2a and $c2c_{cvg}$'s implementations provided by the developers of

each technique. For TurboMQ, we used our own implementation based on the technique described by the original authors [179].

MoJoFM [253] is defined by the following formula,

$$MoJoFM(M) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\% \tag{5.1}$$

where $mno(A, B)$ is the minimum number of Move or Join operations needed to transform the recovered architecture A into the ground truth B. This measure allows us to compare the architecture recovered by the different techniques according to their similarity with the ground-truth architecture. A score of 100% indicates that the architecture recovered is the same as the ground-truth architecture. A lower score results in greater disparity between A and B. MoJoFM has been shown to be more accurate than other measures and was used in the latest empirical study of architecture recovery techniques [64, 111].

Architecture-to-architecture [122] (a2a) is designed to address some of MoJoFM drawbacks. MoJoFM's Join operation is excessively cheap for clusters containing a high number of elements. This is particularly visible for large projects. This results in high MoJoFM values for architectures with many small clusters. In addition, we discovered that MoJoFM does not properly handle discrepancy of files between the recovered architecture and the ground truth. This observation corroborates results obtained in recent work [122]. We tried to reduce this problem by adding the missing files to the recovered architecture into a separate cluster before measuring MoJoFM, but this does not entirely solve the issue. In complement of MoJoFM, we use a new metric, a2a, based on architecture adaptation operations identified in previous work [170, 193]. a2a is a distance measure between two architectures:

$$a2a(A_i, A_j) = (1 - \frac{mto(A_i, A_j)}{aco(A_i) + aco(A_j)}) \times 100\%$$

$$\begin{aligned} mto(A_i, A_j) = {} & remC(A_i, A_j) + addC(A_i, A_j) + \\ & remE(A_i, A_j) + addE(A_i, A_j) + movE(A_i, A_j) \\ aco(A_i) = {} & addC(A_\emptyset, A_i) + addE(A_\emptyset, A_i) + movE(A_\emptyset, A_i) \end{aligned}$$

where $mto(A_i, A_j)$ is the minimum number of operations needed to transform architecture A_i into A_j; and $aco(A_i)$ is the number of operations needed to construct architecture A_i from a "null" architecture $A_\emptyset$.

mto and *aco* are used to calculate the total numbers of the five operations used to transform one architecture into another: additions (*addE*), removals (*remE*), and moves (*movE*) of implementation-level entities from one cluster (i.e., component) to another; as well as additions (*addC*) and removals (*remC*) of clusters themselves. $mto(A_i, A_j)$ is calculated optimally by using the Hungarian algorithm [185] to maximise the weights of a bipartite graph built with the clusters from A_i and A_j.

Cluster-to-cluster coverage ($c2c_{cvg}$) is a metric used in our previous work [66] to assess component-level accuracy. This metric measures the degree of overlap between the implementation-level entities contained in two clusters:

$$c2c(c_i, c_j) = \frac{|entities(c_i) \cap entities(c_j)|}{max(|entities(c_i)|, |entities(c_j)|)} \times 100\%$$

where c_i is a technique's cluster; c_j is a ground-truth cluster; and $entities(c)$ is the set of entities in cluster c. The denominator is used to normalize the entity overlap in the numerator by the number of entities in the larger of the two clusters. This ensures that *c2c* provides the most conservative value of similarity between two clusters.

To summarize the extent to which clusters of techniques match ground-truth clusters, we leverage *architecture coverage* ($c2c_{cvg}$). $c2c_{cvg}$ is a change metric from our previous work [66] that indicates the extent to which one architecture's clusters overlap the clusters of another architecture:

$$c2c_{cvg}(A_1, A_2) = \frac{|simC(A_1, A_2)|}{|A_2.C|} \times 100\%$$

$$simC(A_1, A_2) = \{c_i \mid \begin{array}{l}(c_i \in A_1, \exists c_j \in A_2) \wedge \\ (c2c(c_i, c_j) > th_{cvg})\}\end{array}$$

A_1 is the recovered architecture; A_2 is a ground-truth architecture; and $A_2.C$ are the clusters of A_2. th_{cvg} is a threshold indicating how high the *c2c* value must be for a technique's cluster and a ground-truth cluster in order to count the latter as covered.

Normalized Turbo Modularization Quality (normalized TurboMQ) is the final metric we are using in this work. Modularization metrics measure the quality of the organization and cohesion of clusters based on the dependencies. They are widely accepted metrics which have been used in several studies [15,162,180]. We implemented the TurboMQ version because it has better performance than *BasicMQ* [179].

To compute TurboMQ two elements are required: intra-connectivity, and extra-connectivity. The assumption behind this metric is that architectures with high intra-connectivity are preferable to architectures with a lower intra-connectivity. For each cluster, we calculate a Cluster Factor as followed:

$$CF_i = \frac{\mu_i}{\mu_i + 0.5 \times \sum_j \epsilon_{ij} + \epsilon_{ji}}$$

μ_i is the number of intra-relationships;ϵij + ϵ_{ji} is the number of inter-relationships between cluster i and cluster j. TurboMQ is defined as the sum of all the Cluster Factors:

$$TurboMQ = \sum_{i=1}^{k} CF_i$$

We note that TurboMQ by itself is biased toward architectures with a large number of clusters because the sum of CF_i will be very high if the recovered architecture contains numerous clusters. Indeed, we found that for Chromium, the architecture recovered by ACDC contains thousands of clusters. The TurboMQ value for this architecture was 400 times higher than the TurboMQ values of architectures obtained with other recovery techniques. To address this issue, we normalized TurboMQ by the number of clusters in the recovered architecture.

5.5 Evaluation and Results

This section presents the results of our study that answer the three research questions, followed by a comparison of our results and those of prior work. Tables 5.2-5.21 show the results for all four metrics when applied to a combination of a recovery technique and system; and, if applicable for such a combination, the results for a type of dependency: *Inc*lude, *Sym*bol, *Funct*ion alone, function and global variables (*F-GV*), *Trans*itive, and *Mod*ule-level dependencies. Symbol dependencies may be resolved by ignoring dynamic bindings (No Vir) or using a class hierarchy analysis of dynamic bindings (*S-CHA*) or with interface-only resolution of dynamic bindings (*S-Int*). Bash does not contain dynamic bindings because it is implemented in C, and our tool cannot extract function pointers.

For certain combinations of recovery techniques and systems, a result may not be attainable due to inapplicable combinations (NA), techniques running out of memory (MEM), or timing out (TO). For example, information retrieval-based techniques such as ARC and ZBR do not rely on dependencies. Therefore, normalized TurboMQ results are

Algo.	Bash				
	Inc.	**Sym.**	**Trans.**	**Funct.**	**F-GV**
ACDC	52	57	38	49	50
Bunch-NAHC	53	43	34	49	46
Bunch-SAHC	57	52	34	43	49
WCA-UE	34	24	24	29	30
WCA-UENM	34	24	24	31	30
LIMBO	34	27	27	22	22
K-means	59	55	49	47	46
ARC	43				
ZBR-tok	41				
ZBR-uni	29				
Dir. Struc.	57				

Table 5.2: MoJoFM results for Bash.

Algo.	Chromium						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Mod.**	**Funct.**	**F-GV**
ACDC	64	70	73	71	62	71	71
Bunch-NAHC	28	31	24	29	52	29	35
Bunch-SAHC	12†	71†	43†	42	57	39	29
WCA-UE	23	23	23	27	76	29	29
WCA-UENM	23	23	23	27	73	29	29
LIMBO	TO	23	3	26	79	27	27
K-means	40	42	43	43	78	45	45
ARC	54						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	69						

Table 5.3: MoJoFM results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

Algo.	ITK						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Mod.**	**Funct.**	**F-GV**
ACDC	52	55	52	48	35	60	60
B.-NAHC	37	36	35	35	46	45	47
B.-SAHC	32	46	43	41	51	54	53
WCA-UE	30	31	44	45	64	36	36
WCA-UENM	30	31	44	45	61	36	36
LIMBO	30	31	44	38	60	36	35
K-means	38	42	39	43	68	60	61
ARC	24						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	59						

Table 5.4: MoJoFM results for ITK.

Algo.	ArchStudio						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Trans.**	**Funct.**	**F-GV**
ACDC	60	60	77	78	71	75	74
Bunch-NAHC	48	40	49	47	40	53	46
Bunch-SAHC	54	39	53	40	38	53	54
WCA-UE	30	30	32	45	32	31	31
WCA-UENM	30	30	32	45	33	31	31
LIMBO	23	23	24	25	24	24	23
K-means	44	37	39	41	43	39	38
ARC	56						
ZBR-tok	48						
ZBR-uni	48						
Dir. Struc.	88						

Table 5.5: MoJoFM results for ArchStudio.

Algo.	Hadoop						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Trans.**	**Funct.**	**F-GV**
ACDC	24	29	41	41	28	41	41
Bunch-NAHC	23	21	24	24	17	26	26
Bunch-SAHC	24	26	28	26	20	29	28
WCA-UE	13	12	15	28	17	17	17
WCA-UENM	13	12	15	28	17	17	17
LIMBO	15	13	14	14	13	13	14
K-means	30	25	29	28	29	29	29
ARC	35						
ZBR-tok	29						
ZBR-uni	38						
Dir. Struc.	63						

Table 5.6: MoJoFM results for Hadoop.

Algo.	Bash				
	Inc.	**Sym.**	**Trans.**	**Funct.**	**F-GV**
ACDC	65	80	80	41	41
Bunch-NAHC	68	84	83	41	41
Bunch-SAHC	69	84	83	40	41
WCA-UE	65	81	81	40	40
WCA-UENM	65	81	81	39	40
LIMBO	63	79	79	38	37
K-means	67	84	84	41	40
ARC	67				
ZBR-tok	71				
ZBR-uni	70				
Dir. Struc.	64				

Table 5.7: a2a results for Bash.

Algo.	ITK						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Mod.**	**Funct.**	**F-GV**
ACDC	67	74	63	58	84	48	48
Bunch-NAHC	71	78	68	58	85	47	47
Bunch-SAHC	69	78	66	57	85	48	47
WCA-UE	74	82	47	39	89	48	48
WCA-UENM	74	82	47	39	88	48	48
LIMBO	70	78	44	36	87	46	46
K-means	74	82	71	61	89	51	51
ARC	54						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	61						

Table 5.8: a2a results for ITK.

Algo.	Chromium						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Mod.**	**Funct.**	**F-GV**
ACDC	71	73	74	64	82	62	62
Bunch-NAHC	69	73	76	66	81	63	63
Bunch-SAHC	60†	71†	66†	66	83	64	62
WCA-UE	70	75	78	68	84	66	66
WCA-UENM	70	75	78	68	82	66	66
LIMBO	TO	70	73	64	83	61	61
K-means	71	74	77	67	86	65	65
ARC	54						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	60						

Table 5.9: a2a results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

Algo.	ArchStudio						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Trans.**	**Funct.**	**F-GV**
ACDC	71	86	88	83	92	87	88
Bunch-NAHC	69	80	81	75	80	81	81
Bunch-SAHC	70	80	82	74	80	81	82
WCA-UE	70	83	84	81	83	82	83
WCA-UENM	70	83	84	81	84	82	83
LIMBO	67	79	79	74	78	77	78
K-means	70	81	82	77	83	81	82
ARC	84						
ZBR-tok	85						
ZBR-uni	86						
Dir. Struc.	87						

Table 5.10: a2a results for ArchStudio.

Algo.	Hadoop						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Trans.**	**Funct.**	**F-GV**
ACDC	68	81	84	79	80	84	84
Bunch-NAHC	67	79	80	76	78	80	80
Bunch-SAHC	67	80	81	76	79	81	81
WCA-UE	68	80	80	78	81	81	81
WCA-UENM	68	80	80	78	81	81	81
LIMBO	67	79	79	75	79	78	79
K-means	70	81	81	77	82	82	82
ARC	82						
ZBR-tok	81						
ZBR-uni	83						
Dir. Struc.	88						

Table 5.11: a2a results for Hadoop.

Algo.	Bash				
	Inc.	**Sym.**	**Trans.**	**Funct.**	**F-GV**
ACDC	9	22	6	29	29
Bunch-NAHC	25	31	20	33	28
Bunch-SAHC	30	30	20	28	28
WCA-UE	0	7	7	10	10
WCA-UENM	0	7	7	5	10
LIMBO	6	13	8	7	7
K-means	0	17	6	14	16
ARC	5	11	NA		
ZBR-tok	2	8	NA		
ZBR-uni	2	5	NA		
Dir. Struc.	1	4	NA		

Table 5.12: Normalized TurboMQ results for Bash.

Algo.	Chromium						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Mod.**	**Funct.**	**F-GV**
ACDC	15	19	18	20	46	24	24
Bunch-NAHC	4	24	9	26	51	16	19
Bunch-SAHC	$2^{\dagger}$	$30^{\dagger}$	$11^{\dagger}$	23	45	29	11
WCA-UE	0	2	2	2	36	2	2
WCA-UENM	0	2	2	2	37	2	3
LIMBO	TO	2	2	2	34	2	2
K-means	0	17	13	19	35	22	22
ARC	2	5	NA				
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	2	5	NA				

Table 5.13: Normalized TurboMQ results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

Algo.	ITK						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Mod.**	**Funct.**	**F-GV**
ACDC	33	24	18	32	47	40	40
Bunch-NAHC	15	23	23	22	50	34	37
Bunch-SAHC	10	29	23	21	50	44	37
WCA-UE	3	9	3	2	51	11	9
WCA-UENM	3	9	3	2	47	10	9
LIMBO	7	11	5	1	35	9	9
K-Means	13	24	15	13	37	31	25
ARC	9	33	NA				
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	9	9	NA				

Table 5.14: Normalized TurboMQ results for ITK.

Algo.	ArchStudio						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Trans.**	**Funct.**	**F-GV**
ACDC	66	41	76	84	71	72	74
Bunch-NAHC	72	42	74	85	35	74	75
Bunch-SAHC	71	41	76	85	50	72	74
WCA-UE	1	11	22	65	15	10	19
WCA-UENM	1	11	22	65	15	10	19
LIMBO	2	12	31	38	7	24	27
K-means	13	21	38	51	29	35	39
ARC	18	25	NA				
ZBR-tok	6	17	NA				
ZBR-uni	5	15	NA				
Dir. Struc.	1	26	NA				

Table 5.15: Normalized TurboMQ results for ArchStudio.

Algo.	Hadoop						
	Inc.	**S-CHA**	**S-Int**	**No DyB**	**Trans.**	**Funct.**	**F-GV**
ACDC	48	28	59	65	29	57	58
Bunch-NAHC	40	26	53	61	17	52	48
Bunch-SAHC	40	31	53	61	18	54	56
WCA-UE	1	5	8	34	7	6	8
WCA-UENM	1	5	8	33	7	6	8
LIMBO	2	7	19	25	2	17	17
K-means	11	13	29	34	9	26	27
ARC	6	13	NA				
ZBR-tok	5	10	NA				
ZBR-uni	7	13	NA				
Dir. Struc.	1	20	NA				

Table 5.16: Normalized TurboMQ results for Hadoop.

Algo.	Bash														
	Inc.			**Sym.**			**Trans.**			**Funct.**			**F-GV**		
ACDC	20	47	71	27	77	93	13	40	94	0	7	50	0	7	21
Bunch-NAHC	6	20	53	6	20	71	1	10	46	1	10	36	1	9	37
Bunch-SAHC	7	23	43	16	37	81	3	13	46	0	6	41	0	11	37
WCA-UE	0	6	63	0	4	61	0	1	58	0	0	26	0	1	23
WCA-UENM	0	6	63	0	4	61	0	1	58	0	2	24	0	6	45
LIMBO	0	0	57	0	0	60	0	0	63	0	0	32	0	0	32
K-means	10	19	49	9	28	69	9	26	57	0	5	41	0	2	46
ARC	4			20						54					
ZBR-tok	7			7						71					
ZBR-uni	0			0						50					
Dir. Struct.	14			36						64					

Table 5.17: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Bash.

Algo.	ITK																				
	Inc.			S-CHA			S-Int			No DyB			Mod.			Funct.			F-GV		
ACDC	0	0	57	0	0	48	0	0	31	0	0	31	8	8	54	0	8	38	0	8	38
B.-NAHC	0	0	31	0	0	38	0	0	34	0	0	34	5	23	60	0	0	32	0	0	31
B.-SAHC	0	0	0	0	2	54	0	0	29	0	0	23	6	29	57	0	0	34	0	0	32
WCA-UE	0	0	23	0	0	24	0	8	23	0	0	8	18	40	78	0	0	38	0	0	38
WCA-UENM	0	0	23	0	0	23	0	8	23	0	0	8	18	32	80	0	0	38	0	0	38
LIMBO	0	0	8	0	0	12	0	0	5	0	0	0	26	40	86	0	0	9	0	0	9
K-means	0	0	44	0	8	51	0	3	53	0	9	54	29	46	87	0	6	45	0	8	53
ARC	0						0						23								
ZBR-tok	MEM																				
ZBR-uni	MEM																				
Dir. Struct.	0						0						8								

Table 5.18: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ITK.

Algo.	Chromium																				
	Inc.			S-CHA			S-Int			No DyB			Mod.			Funct.			F-GV		
ACDC	16	30	80	17	45	92	17	37	87	10	23	83	13	17	28	9	17	80	10	17	80
B.-NAHC	0	0	7	0	0	26	0	0	3	0	0	9	7	14	33	0	0	4	0	3	25
B.-SAHC	0	6	19	14	33	80	7	12	36	4	10	54	8	16	39	0	1	38	0	0	4
WCA-UE	0	0	3	0	0	3	0	0	3	0	0	3	17	37	79	0	0	3	0	0	4
WCA-NM	0	0	3	0	0	3	0	0	3	0	0	3	12	35	76	0	0	3	0	0	4
LIMBO	TO			0	0	0	0	0	0	0	0	0	25	45	69	0	0	0	0	0	0
K-means	3	7	31	1	6	25	2	8	24	1	5	20	24	43	78	1	5	21	1	4	20
ARC	1						3						25								
ZBR-tok	MEM																				
ZBR-uni	MEM																				
Dir. Struct.	4						7						23								

Table 5.19: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

Algo.	ArchStudio																			
	Inc.			S-CHA			S-Int			No DyB			Trans.			Funct.			F-GV		
ACDC	9	21	47	21	54	77	56	77	93	52	75	89	52	72	85	42	64	77	42	62	77
B.-NAHC	2	5	21	2	6	28	3	11	41	3	10	34	2	5	26	5	13	44	3	9	41
B.-SAHC	3	10	35	2	7	33	5	13	46	5	8	20	4	7	27	3	11	48	5	19	46
WCA-UE	0	5	37	0	5	29	2	14	38	19	35	53	2	13	39	0	7	37	0	8	47
WCA-NM	0	5	37	0	5	29	2	14	38	19	35	53	2	13	39	0	7	37	0	8	47
LIMBO	0	0	69	0	0	65	0	0	68	0	0	74	0	0	63	0	0	66	0	0	68
K-means	6	25	58	1	18	57	5	24	53	8	23	53	8	33	69	4	22	53	5	24	57
ARC	9									29						59					
ZBR-tok	4									16						65					
ZBR-uni	4									23						47					
Dir. Struct.	74									91						100					

Table 5.20: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ArchStudio.

Algo.	Hadoop																			
	Inc.			S-CHA			S-Int			No DyB			Trans.			Funct.			F-GV		
ACDC	0	3	43	4	13	39	7	18	49	7	18	45	4	10	36	7	16	52	9	16	52
B.-NAHC	1	3	35	1	1	28	1	4	36	1	4	33	1	3	24	2	5	35	2	5	38
B.-SAHC	1	3	32	1	7	40	2	6	35	2	6	34	1	3	20	2	8	38	1	5	36
WCA-UE	0	7	37	0	12	29	1	12	33	2	9	42	1	12	33	2	15	38	1	12	35
WCA-NM	0	7	37	0	12	29	1	12	33	2	9	41	1	12	33	2	15	38	1	12	35
LIMBO	0	0	64	0	0	54	0	0	55	0	0	64	0	0	58	0	0	57	0	0	55
K-means	2	22	71	1	15	60	3	19	64	2	16	60	1	14	53	3	18	65	3	19	64
ARC	6									24						63					
ZBR-tok	4									16						65					
ZBR-uni	4									23						47					
Dir. Struct.	28									45						75					

Table 5.21: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Hadoop.

not meaningful when studying the impact of the different factors of the dependencies. For this reason, we only report normalized TurboMQ for include and symbol dependencies and mark the other combinations as inapplicable.

We do not report results obtained utilizing transitive dependencies for Chromium and ITK because, as discussed above, the use of such dependencies with those projects caused scalability problems. Module-level dependencies are only reported for ITK and Chromium, since they are the only projects that define modules in their documentation or configuration files.

5.5.1 RQ1: Can accurate dependencies improve the accuracy of recovery techniques?

As explained in section 5.2.1, include dependencies present some issues (e.g. missing relationships between non-header files, etc.) which can be solved by using more accurate dependencies based on symbol interactions. Therefore, to answer this research question, we focus on results obtained using include (Inc) and symbol dependencies (Sym, S-Int, and S-CHA), which are presented in Tables 5.2-5.21. In these Tables, we reported the average results for each technique and each type of dependency.

Three recovery techniques—ARC, ZBR-tok, and ZBR-uni—do not rely on dependencies; however, we include them to assess the accuracy of symbol dependencies against these information retrieval-based techniques. The best score obtained for each recovery technique across all type of dependencies is highlighted in dark gray; the best score between include and symbol dependencies for each technique, when applied to a particular technique, is highlighted in light gray.

Our results indicate that symbol dependencies generally improve the accuracy of recovery techniques over include dependencies. According to a2a scores (Tables 5.7-5.11) relying on both types of symbol dependencies outperforms relying on include dependencies for all of the combinations of techniques and systems which use dependencies. The only exception is in the case of ITK, where relying on include dependencies outperforms interface-only resolution for dynamic bindings. As ITK contains a large number of dynamic-bindings dependencies (more than 75%), using interface-only resolution likely results in a significant loss of information, making those dependencies inaccurate. When doing a complete analysis of the dynamic-bindings dependencies of ITK (S-CHA), using symbol dependencies with a class hierarchy analysis of dynamic bindings outperforms using include dependencies for all techniques. On average, using symbol dependencies respectively improves the accuracy by 9 percentage points (percentage point, pp, is the unit for the arithmetic difference

between two percentages) according to a2a. For a2a, the technique obtaining the greatest improvement from the use of symbol dependencies, as compared to include dependencies, is K-means, followed by Bunch-SAHC, with an average improvement of, respectively, 12 pp and a 10 pp for a2a.

MoJoFM results (Tables 5.2 to 5.16) followed a similar trend, with symbol dependencies generally improving the accuracy of the recovered architecture over include dependencies for five of the projects. However, for Bash, include dependencies produce better results than symbol dependencies for all techniques but ACDC.

Tables 5.17-5.21 show $c2c_{cvg}$ for three different values of th_{cvg}, i.e., 50%, 33%, and 10%, (from left to right) for each combination of technique and dependency type. The first value depicts $c2c_{cvg}$ for $th_{cvg} = 50\%$ which we refer to as a *majority* match. We select this threshold to determine the extent to which clusters produced by techniques mostly resemble clusters in the ground truth. The other two $c2c_{cvg}$ scores show the portion of *moderate* matches (33%) and *weak* matches (10%).

Dark gray cells show the highest $c2c_{cvg}$ for each recovery technique across all type of dependencies. Light gray cells show the highest $c2c_{cvg}$ between include and symbol dependencies for each technique, when applied to a particular technique for a specific threshold th_{cvg}. Several rows do not have any highlighted cells; such rows indicate that $c2c_{cvg}$ is identical for include and symbol dependencies. We observe significant improvement when using symbol dependencies over include dependencies, even for $th_{cvg} = 50\%$. For example, in Table 5.20, for ACDC on ArchStudio, the $c2c_{cvg}$ for $th_{cvg} = 50\%$ for include dependencies is 9%, while using symbol dependencies increased it to 56% with symbol dependencies and interface-only resolution. Overall, Tables 5.17 to 5.21 indicate that (1) the use of symbol dependencies generally produces more accurate clusters (majority matches); and that (2) $c2c_{cvg}$ is low regardless of the types of dependencies used.

Tables 5.12 to 5.16 presents the normalized TurboMQ results, which measure the organization and cohesion of clusters independent of ground-truth architectures. Both types of symbol dependencies generally obtain higher normalized TurboMQ scores than include dependencies, ACDC and Bunch with CHA resolution being exceptions for ArchStudio and Hadoop. In other words, symbol dependencies help recovery techniques produce architectures with better organization and internal component cohesion than include dependencies. TurboMQ results of the summation of individual scores for each cluster in the architecture make it biased toward architectures with an extremely high number of clusters. For example, ACDC for Chromium, with more than 2000 clusters, obtains TurboMQ scores with one to two orders of magnitude larger than the other metrics.

Statistical Significance Test

We conduct statistical significance tests to verify whether using accurate dependencies improves the quality of recovery techniques. We do not use paired t-tests because our data does not follow a normal distribution. Instead, we use the Wilcoxon signed-rank test, which is a non-parametric test.

We also measure the Cliff's δ, a non-parametric effect size metric, to quantify the difference among the different types of dependencies. Cliff's δ ranges from -1 to 1. The sign of the δ indicates whether the first or second sample contains higher values. For example, when looking at results using include versus results obtained using symbol dependencies with CHA resolution for MoJoFM in Table 5.23, the δ is positive, indicating that results obtained with symbol dependencies and CHA resolution are generally better than the ones obtained using include dependencies results. In contrast, the δ for direct versus transitive dependencies is negative, indicating that using direct dependencies generally produces higher results. To interpret the effect size, we use the following magnitude: negligible ($|\delta| < 0.147$), small ($|\delta| < 0.33$), medium ($|\delta| < 0.474$), and large ($0.474 \leq |\delta|$) [204]. To compute these tests, we use average measurements obtained for each type of dependencies and metrics across all projects. This represents 35 paired samples per metric. We report these results in Tables 5.22 and 5.23.

When comparing include dependencies and symbol dependencies with interface-only dynamic-bindings resolution, for 4 out of 6 metrics—a2a, TurboMQ, and two $c2c_{cvg}$ metrics (majority and moderate matches)— we found that the p-values are inferior to 0.001, suggesting our results are statistically significant. For these four metrics, we observed a reasonably high effect size, varying from small ($c2c_{cvg}$ metrics) to large (a2a).

When comparing include dependencies and symbol dependencies with CHA, we found that the p-values are inferior to 0.02, for 3 out of 6 metrics. The effect size for these 3 metrics vary from negligible ($c2c_{cvg}$ metrics) to large (a2a).

$c2c_{cvg}$ with weak coverage not being significant indicates that the type of dependency does not matter for producing a "weak" approximation of the ground-truth architecture. However, when attempting to obtain a better architecture (i.e. with a moderate coverage), working with symbol dependencies with interface-only dynamic-bindings resolution is preferable.

Table 5.22: Wilcoxon Signed Rank when comparing different types of dependencies.

Metrics	p-values				
	Inc. vs S-CHA	**Inc. vs S-Int**	**S-Int vs Trans.**	**S-Int vs Funct.**	**Funct. vs F-GV**
MoJoFM	.87	.06	.004	.60	.59
a2a	<.001	<.001	.61	<.001	.32
TurboMQ	.03	<.001	<.001	.28	.28
$c2c_{cvg}$ _weak	.20	.10	.02	.007	.77
$c2c_{cvg}$ _mod	.02	<.001	.02	.006	.82
$c2c_{cvg}$ _maj	.22	.003	.01	.24	.64

Table 5.23: Cliff's δ Effect Size Tests. A negative value indicates that the effect size favor of the first dependency type listed.

Metrics	Effect Size				
	Inc. vs S-CHA	**Inc. vs S-Int**	**S-Int vs Trans.**	**S-Int vs Funct.**	**Funct. vs F-GV**
MoJoFM	.03	.11	-.19	.05	-.01
a2a	.96	.64	-.10	-.37	.02
TurboMQ	.36	.39	-.48	.08	.01
$c2c_{cvg}$ _weak	.12	.10	-.14	-.20	-.01
$c2c_{cvg}$ _mod	.09	.17	-.14	-.17	.03
$c2c_{cvg}$ _maj	.05	.17	-.23	-.11	-.006

Summary of RQ1

The overall conclusion from applying these four metrics is that symbol dependencies allow recovery techniques to increase their accuracy for all systems in almost every case, independently of the metric chosen. Especially, using a2a metrics, we observed a statistically significant improvement coupled with a large effect size in favor of symbol dependencies with interface-only dynamic-bindings.

Despite the accuracy improvement of using symbol dependencies over include dependencies, $c2c_{cvg}$ results for majority match are low. This indicates that these techniques' clusters are significantly different from clusters in the corresponding ground truth. It suggests that improvement is needed for all the evaluated recovery techniques.

5.5.2 RQ2: What is the impact of different input factors, such as the granularity level, the use of transitive dependencies, the use of different symbol dependencies and dynamic-bindings graph construction algorithms, on existing architecture recovery techniques?

There are several types of dependencies that can be used as input to recovery techniques. First, we can break symbol dependencies based on the type of symbol (functions, global variables, etc.). Another important factor to take into consideration concerns the way we resolve dynamic bindings. Finally, we also look at the transitivity and the granularity level of the dependencies. Those different factors are considered as important for other software analyzes and have a significant impact on the quality of the recovered architectures.

Impact of Function Calls and Global Variables

Function calls are the most common type of dependencies in a system. The question we study pertains to whether the most common dependencies have the most significant impact on the quality of the recovery techniques.

Global variables typically represent a small, but important part of a program. Global variables are a convenient way to share information between different functions of the same program. If two functions use the same global variables, they might be similar and the files they belong to could be clustered together. Two functions may, in fact, be dependent on each other if they both utilize the same global variable. Therefore, adding a global-variable-usage graph to the function-call graph could help connect similar elements, that do not directly interact with one another via function calls.

We do not use global variable dependencies alone, because for most of the systems, only a minority of elements accesses global variables. Therefore, by considering global variable dependencies alone, we would miss a large number of elements of the system, making several metrics inaccurate. Instead, we measure the improvement on the quality of the recovered architectures obtained by adding global variable dependencies to functions calls compared to using function calls alone. Static analysis using LLVM can detect global variables for C and C++ projects. For Java projects, we consider variables containing the **static** keyword as an approximation of C/C++ global variables.

Overall, MoJoFM and normalized TurboMQ values for function calls alone and symbol dependencies are highly similar. However, for $c2c_{cvg}$ and a2a, results from all symbol

dependencies are significantly better than results from function calls alone. For example, a2a results are, on average 16.8 pp better when using all symbol dependencies available (S-CHA) than when using function calls alone. Despite the fact that function calls have a major impact on the accuracy of architecture recovery techniques, using function calls alone is not sufficient for obtaining accurate recovered architectures.

The impact of global variable usage is minor. For example, on average, adding global variable usage to function call dependencies improves the results by 0.1 pp according to a2a. The impact of global variables is reduced because of the small number of global variable accesses in the projects used in our study. For example, Chromium's C and C++ Style Guide [3] discourages the use of global variables. We acknowledge that our results would likely be different for a system relying heavily on global variables, such as the Toyota ETCS system, which contains about 11,000 global variables [110, 114].

We performed statistical tests and confirmed that symbol dependencies are better than functions alone for architecture recovery for 3 out of 6 metrics (p-value <0.05), with an effect size varying from small to medium in favor of symbol dependencies with interface-only resolution. In addition, we did not observe a statistical difference among architectures recovered from dependencies involving functions alone, and dependencies involving both functions and global variables. It confirms that global variable usage is not a key dependency for accurate architecture recovery for the projects we studied.

Summary: The impact of global variable on the quality of the recovered architectures is minor and using symbol dependencies produces better architectures than using function calls alone.

Impact of Dynamic-bindings Resolution

Dynamic-bindings resolution is a known problem in software engineering, and several possible strategies for addressing it have been proposed [18, 46, 227]. Due to the high number of dynamic bindings in C++ and Java projects, the type of resolution chosen when extracting symbol dependencies can significantly impact the accuracy of recovery techniques. However, it is unclear which type of dynamic-bindings resolution has the greatest impact on the architecture of a system. To determine which dynamic-bindings resolution to utilize for recovery techniques, we evaluate three different resolution strategies: (1) ignoring dynamic bindings, (2) interface-only resolution, and (3) CHA-based resolution.

[3] https://www.chromium.org/developers/coding-style

Ignoring dynamic bindings is the easiest solution to follow. More importantly, including it as a possible resolution strategy allows us to determine whether doing any dynamic bindings analysis improves recovery results.

For interface-only resolution, we only consider the interface of the virtual functions as being invoked and discard potential calls to derived functions. This is the simplest resolution that can be performed that does not ignore dynamic bindings.

For the third resolution strategy, we use Class Hierarchy Analysis (CHA) [46], which is a well-known analysis that is computationally costly to perform. For this type of resolution, we consider all the derived functions as potential calls. This resolution also creates a larger dependency graph than interface-only resolution.

The results obtained when ignoring dynamic bindings are shown in column No DyB. The results for symbol dependencies obtained with CHA and Interface-only dynamic-bindings resolution are respectively presented in column S-CHA and S-Int. Bash, written in C, is the only project which does not contain any dynamic bindings.

The results obtained when discarding dynamic bindings (column No DyB) are generally not as good as with other symbol dependencies. According to a2a, using only non-dynamic-bindings dependencies reduces the accuracy of the recovery techniques for all projects and all techniques when compared to using dynamic bindings and the average a2a results without dynamic bindings are 11 pp lower than the results with CHA and 6 pp lower than the results with interface-only resolution.

There are a few exceptions for Chromium, Hadoop, and ArchStudio with the metrics MoJoFM and normalized TurboMQ. The reason for unexpectedly high results with MoJoFM and normalized TurboMQ is that using partial symbol dependencies is not well handled by those two metrics. Using partial symbol dependencies —in our case, we discard symbol dependencies that are dynamic bindings— results in (1) a significant mismatch of files between the ground-truth architecture and the recovered architectures, and (2) a disconnected dependency graph. The file mismatches create artificially high MoJoFM results, and the disconnected dependency graphs can lead to extremely high or even perfect normalized TurboMQ scores, as it is the case for ArchStudio when using Bunch without dynamic-bindings dependencies. $c2c_{cvg}$ results are not conclusive either way.

When looking at interface-only and CHA resolutions, we observe a difference in behavior of the two Java projects and the two C/C++ projects. For Java-based Hadoop and ArchStudio, using an interface-only resolution seems to greatly improve the results over using CHA. Those results are obtained for both projects and for all metrics, with only two exceptions for $c2c_{cvg}$ in Hadoop (Table 5.20) where using CHA provides slightly better results. On average, according to normalized TurboMQ, using interface-only resolution

improves the results by 20 pp for ArchStudio and Hadoop. However, for C++-based ITK and Chromium, the normalized TurboMQ results are improved by 6 pp when using CHA for dynamic-bindings resolution.

There could be several reasons for this difference. First, the two C++ projects are between 10 and 200 times larger than the two Java projects we studied. It is possible that a complete analysis of dynamic bindings only becomes necessary for large projects with many complex virtual objects. Second, in Java, methods are virtual by default, while in C++, methods have to be declared as virtual by using the keyword `virtual`. C and C++ developers also have the possibility to use function pointers instead of dynamic bindings, which are currently not handled properly by our symbol dependency extractor. Those two elements could also be a reason why we observed different affects of dynamic-bindings resolutions for C++ and Java projects.

Summary: Our overall results indicate that, to obtain a more accurate recovered architecture, the choice of the dynamic-bindings resolution algorithm depends on the project studied. Specifically, if the project contains a high number of dynamic bindings, CHA is likely to produce better recovery results. Otherwise, interface-only resolution is preferable. Ignoring dynamic bindings is ill-advised in most cases.

Transitive vs. Direct Dependencies

A transitive dependency can be built from direct dependencies. For example, if A depends on B, and B depends on C, then A transitively depends on C. Recovery techniques can use as input (1) direct dependencies only or (2) transitive dependencies. To compare direct dependencies against transitive dependencies, we run a transitive closure algorithm on the symbol dependencies and study the effect of adding transitive dependencies on the accuracy of architecture recovery. We did not use include dependencies for this study because, as explained in Section 5.2.1, include dependencies for C and C++ projects are not direct dependencies. Furthermore, we did not include Chromium because the algorithm generating transitive dependencies does not scale to that size, even when we tried to use advanced computational techniques, such as Crocopat's use of binary decision diagrams [21]. For ITK, although we were able to obtain transitive dependencies, none of the architecture recovery techniques scaled to its size. Therefore, we cannot report those results. Results for Bash, Hadoop, and ArchStudio, are reported in the tables corresponding to the different metrics (column Trans).

When comparing the results obtained with direct (Sym for Bash and S-Int for Hadoop and ArchStudio) and transitive (Trans) symbol dependencies, we observe that using direct

dependencies generally provides similar or better results. Results with MoJoFM, normalized TurboMQ, and $c2c_{cvg}$ tend to favor the use of direct dependencies over transitive dependencies (+15 pp on average for normalized TurboMQ when using direct dependencies, +4.9 pp on average for MoJoFM).

According to a2a, using transitive dependencies has a minor impact (-0.33 pp on average) on the results. a2a gives importance to the discrepancy of files between the recovered architecture and the ground truth. As no files are added or removed when obtaining the transitive dependencies from the direct dependencies, this discrepancy is exactly the same between the direct and transitive dependencies. This is why we do not observe a significant difference between direct and transitive dependencies results when using a2a.

When running statistical tests, we found that results from direct dependencies (S-Int) are statistically different from results obtained from transitive dependencies for all metrics, except a2a, confirming our conclusion that direct dependencies have a positive impact on the quality of the recovered architectures.

Summary: With fewer dependencies, using direct dependencies is more scalable than transitive dependencies. In summary, direct dependencies help generate more accurate architectures than transitive dependencies in most cases.

Impact of the Level of Granularity of the Dependencies

Results at the module level are reported for ITK and Chromium under the column Mod. of Tables 5.3, 5.4, 5.8, 5.9, 5.13, 5.14, 5.18, and 5.19. Module dependencies are obtained by adding information extracted from the configuration files to group files together. This information is written by the developers and could represent the architecture of the project as it is understood by developers. Given the inherent architectural information in such dependencies, it is expected that they would improve a recovery technique's accuracy. Because we only have module dependencies for ITK and Chromium, we do not have enough data points to measure statistical significance of our results. However, results obtained from module-level dependencies tend to be much better than from file-level dependencies. For example, on average, compared to using the best file-level dependencies, using module-level information improves the results by 7.5 pp according to a2a.

Summary: Overall, our results indicate that module information, when available, significantly improves recovery accuracy and scalability of all recovery techniques. As shown in Table 5.1, the number of module dependencies is almost 70 times lower than the number of file dependencies. Because of this reduction in the number of dependencies, we obtain results from all recovery techniques in a few seconds when working at the module level, as opposed to several hours for each technique when working at the file level.

5.5.3 RQ3: Can existing architecture recovery techniques scale to large projects comprising 10MSLOC or more?

Scalability

Overall, ACDC is the most scalable technique. It took only 70-120 minutes to run ACDC on Chromium on our server. The WCA variations and ARC have a similar execution time (8 to 14 hours), with WCA-UENM slightly less scalable than WCA-UE. Bunch-NAHC is the last technique which was able to terminate on Chromium for both kinds of dependencies, taking 20 to 24 hours depending on the kind of dependencies used. LIMBO only terminated for symbol dependencies after running for 4 days on our server.

Bunch-SAHC timed out after 24 hours for both include and symbol dependencies. We report here the intermediate architecture recovered at that time. Bunch-SAHC investigates all the neighboring architectures of a current architecture and selects the architecture that improves MQ the most; Bunch-NAHC selects the first neighboring architecture that improves MQ. Bunch-SAHC's investigation of all neighboring architectures makes it less scalable than Bunch-NAHC.

LIMBO failed to terminate for include dependencies after more than 4 days running on our server. Two operations performed by LIMBO, as part of hierarchical clustering, result in scalability issues: construction of new features when clusters are merged and computation of the measure used to compare entities among clusters. Both of these operations are proportional to the size of clusters being compared or merged, which is not the case for other recovery techniques that use hierarchical clustering (e.g., WCA).

ZBR needs to store data of the size nzV, where n is the number of files being clustered, z is the number of zones, and V is the number of terms. For large software (i.e., ITK and Chromium), with thousands of files and millions of terms, ZBR ran out of memory after using more than 40GB of RAM.

The use of symbol dependencies improves the recovery techniques' scalability over include dependencies for large projects (i.e., ITK and Chromium). The main reason for this phenomenon is that include dependencies are less direct than symbol dependencies.

As mentioned in the discussion of the previous research question, working at the module-level significantly reduces the number of dependencies and, therefore, greatly improves the scalability of all dependency-based techniques for large projects. Indeed, at the module-level we were able to obtain results in a few seconds, even for techniques that did not scale with file-level dependencies.

Metrics vs. Size

While some algorithms are scalable for large projects, it does not mean that results obtained for large systems are as relevant as results obtained for smaller systems. We verify if automatic architecture recovery techniques perform equally for software of all sizes by measuring the evolution of the architectures' quality, when the size of the projects increases.

Overall, we can see that results for Chromium (the largest project) are generally less accurate than results for Bash, ArchStudio, or Hadoop (the smallest projects). However, results for ITK are generally worst than for Chromium, despite ITK being ten times smaller. Because we only study 5 different projects, we cannot draw clear conclusions. Nonetheless, the fact that results for ITK are worst than for Chromium seems to indicate that the size of the project under study is not the only factor affecting the quality of recovered architectures. Other factors such as the programming language, the coding style, and the use of dynamic bindings probably also have an impact that we can't measure with only five different projects.

5.5.4 Comparison with Baseline Algorithms

To see whether software architecture recovery algorithms are effective, we compare their results with two baseline recovered architecture.

For the first baseline, we recovered the architecture based on the directory structure of the project. According to a2a results, all recovery techniques performed better than the baseline for Bash, ITK and Chromium. A similar trend can be observed for TurboMQ and $c2c_{cvg}$, with a few exception (e.g. WCA and Limbo for Chromium for TurboMQ). This seems to indicate that architecture recovery techniques might be helpful to improve the architecture of these projects.

Table 5.24: Wilcoxon Signed Rank for each algorithm when compared to K-means

Metrics	p-values					
	ACDC	**Bunch-NAHC**	**Bunch-SAHC**	**WCA-UE**	**WCA-UENM**	**Limbo**
MoJoFM	<.001	.002	.27	<.001	<.001	<.001
a2a	.42	<.001	<.001	.11	.11	<.001
TurboMQ	<.001	<.001	<.001	<.001	<.001	<.001
$c2c_{cvg}$ _weak	.003	<.001	.004	<.001	<.001	0.004
$c2c_{cvg}$ _mod	.002	<.001	.003	<.001	<.001	<.001
$c2c_{cvg}$ _maj	<.001	<.001	.65	<.001	<.001	<.001

Table 5.25: Cliff's δ Effect Size Tests. A negative value indicates that the effect size is in favor of K-means.

Metrics	Effect Size					
	ACDC	**Bunch-NAHC**	**Bunch-SAHC**	**WCA-UE**	**WCA-UENM**	**Limbo**
MoJoFM	.54	-.25	-.04	-.67	-.64	-.86
a2a	-.005	-.15	-.16	-.08	-.08	-.31
TurboMQ	.52	.49	.51	-.66	-.66	-.53
$c2c_{cvg}$ _weak	.27	-.57	-.45	-.56	-.56	-.11
$c2c_{cvg}$ _mod	.24	.58	-.33	-.57	-.55	-.97
$c2c_{cvg}$ _maj	.42	-.25	-.06	-.55	-.55	-.77

For ArchStudio and Hadoop, the directory structure-based architecture consistently outperforms architectures recovered with other algorithms, except for TurboMQ for which results are less consistent. This seems to indicate that ArchStudio and Hadoop already have a directory relatively similar to their ground truth architecture and that architecture refactoring might not be necessary for these two projects.

Our second baseline consists in comparing the algorithms specifically designed for architecture recovery (ACDC, Bunch, WCA, and Limbo) with the results obtained from a basic machine learning algorithm, k-means, used with default parameters. Tables 5.24 and 5.25 show the statistical significance and the effect size of the difference between K-means and other algorithms, independently from the dependencies used. ACDC is the only algorithm that produces equivalent or better results than K-means consistently, for all metrics. WCA-UE and Limbo always produce worst results than K-means. Finally, the two Bunch algorithms produce better results than K-means only for some metrics.

The three techniques that performed consistently worst than the baseline algorithm are all hierarchical clustering algorithms. It is possible that techniques based on hierarchical clustering are not adapted to recover flat architectures. Results could be different for other

projects or ground-truth architectures.

5.5.5 Summary of Results

Overall, we discovered three main findings from our study. First, using accurate symbol dependencies improves the quality of the recovered architectures. Second, using direct dependencies is more scalable and generally improves the quality of the results. Finally, working with high-level dependencies (i.e. module dependencies) is another key factor for scalable and high-quality architecture recovery of large systems.

5.5.6 Comparison with the Prior Work

As previously mentioned, three of our subject systems were also used in our previous study [64]. It is difficult to compare our results with the prior study because of the differences described in Section 5.4.3. When using the same type of dependencies (Inc) as in our previous study, we observe minor differences for some algorithms. However, on average, the MoJoFM scores only drop by 0.1 pp for all techniques over the scores reported in [64]. In the cases of Hadoop and ArchStudio, our previous study used a different level of granularity (class level), which makes comparison with current work irrelevant.

5.6 Threats to Validity

This section describes several secondary results of our research such as issues encountered with the different metrics, extreme architectures, and guidelines concerning the dependencies, the architecture recovery techniques, and the metrics to use in future work.

5.6.1 Metrics Limitations

As mentioned in section 5.4.5, some metrics have limitations and can be biased toward specific architectures. In this section, we explain the limitations we encountered with two of the metrics we used. Those limitations appeared because, the metrics in question were neither explicitly intended for nor adapted to specific types of dependencies.

The dependencies are often incomplete. For example, include dependencies generally contain fewer files than the ground-truth architecture. The reasons were explained in

Section 5.2.1, including the fact that non-header-file to non-header-file dependencies are missing. Unfortunately, one of the most commonly used metrics, MoJoFM, assumes that the two architectures under comparison contain the same elements. Given this limitation, one can create a recovery technique that achieves 100% MoJoFM score easily but completely artificially. The technique would simply create a file name that does not exist in a project, and place it in a single-node architecture. The MoJoFM score between the single-node architecture and the ground truth will be 100%. By contrast, the a2a metric is specifically designed to compare architectures containing different sets of elements.

In addition to the "file mismatch" issue with MoJoFM, we also identified issues with TurboMQ, as discussed in Section 5.4.5. Replacing TurboMQ by its normalized version yielded an improvement. However, one has to be careful when using normalized TurboMQ. We identified two boundary cases where normalized TurboMQ results are incorrectly high. It is possible to obtain the maximum score for normalized TurboMQ by grouping all the elements of the recovered architecture in a single cluster. As there will be no inter-cluster dependencies, the score will be 100%. We manually checked all recovered architectures to make sure this specific case never happened in our evaluation. The second "extreme case" occurs when the dependency graph used as input is not fully connected. This can happen when using only partial symbol dependencies (i.e., global variable usage, non-dynamic-bindings dependency graph, etc.). In this case, some recovery techniques will create architectures in which clusters are not connected to one another. This also results in a normalized TurboMQ score of 100%. In our evaluation, this issue occurs when using non-dynamic-bindings dependencies for ArchStudio and Chromium in Tables 5.13 and 5.15. This is a limit of normalized TurboMQ when using partial dependencies.

Those are specific issues we observed performing our analysis. It is conceivable that biases towards other types of architecture have yet to be discovered. This suggests that a separate, more extensive study on the impact of different architectures on the metrics would be useful in order to obtain a better understanding of those metrics. Such a study has not been performed to date.

Metrics are convenient because they quantify the accuracy of an architecture with a score, allowing comparisons between recovery techniques. Our study has included, developed, adapted, and evaluated a larger number of metrics than prior similar studies. However, the value of this score by itself must be treated judiciously. Obviously, the "best" recovered architecture is the one that is the closest to the ground-truth. At the same time, important questions such as "Is the recovered architecture good enough to be used by developers?", "Can an architecture with an a2a score of 90% be used for maintenance tasks?" cannot be answered by solely using metrics. A natural outgrowth of this work, then would be to involve real engineers in performing maintenance tasks in real settings. Then it would

be possible to evaluate the extent to which the metrics are indicative of the impact on completing such tasks. We are currently preparing such a study with the help of several of our industrial collaborators.

The metrics chosen in this work measure the similarity and quality of an architecture at different levels—the system level (measured by MoJoFM and a2a), the component level (measured by $c2c_{cvg}$) and the dependency-cohesion level (measured by normalized TurboMQ). In future work, we intend to measure the accuracy of an architecture from an additional perspective, by analyzing whether the architecture contains undesirable patterns or follows good design principles.

5.6.2 Selecting Metrics and Recovery Techniques

Using only one metric is not enough to assess the quality of architectures. However, some metrics are better than others depending on the context. When working on software evolution, the architectures being compared will likely include a different set of files. In this case, a2a, $c2c_{cvg}$, and normalized TurboMQ are more appropriate than MoJoFM, which assumes that no files are added or removed across versions. If the architectures being compared contain the same files (e.g., comparing different techniques with the same input), a2a will give results with a small range of variations, making it difficult to differentiate the results of each technique. In this case, MoJoFM results are easier to analyze than the ones obtained with a2a.

We do not claim that one recovery technique is better than the others. However, we can provide some guidelines to help practitioners choose the right recovery technique for their specific needs. According to our scalability study, ACDC, ARC, WCA, and Bunch-NAHC are the most adapted to recover large software architectures. When trying to recover the low-level architecture of a system, practitioners should favor ACDC, as it generally produces a high number of small clusters. If a different level of abstraction is needed, WCA, LIMBO, and ARC allow the user to choose the number of clusters of the recovered architecture. Those techniques will be more helpful for developers who already have some knowledge of their project architecture.

5.6.3 Non-uniqueness of Ground-Truth Architectures

There is not necessarily a unique, correct architecture for a system [23, 65]. Recovering ground-truth architectures require heavy manual work from experts. Therefore, it is challenging to obtain different certified architectures for the same system. As we are using

only one ground-truth architecture per project, there is a threat that our study may not reflect other ground-truth architectures. To reduce this threat, we use four different metrics, including one independent of the ground-truth architecture. Two of the metrics used in this study were developed by some authors of this work, which might have caused a bias in this study. However, all four metrics, including metrics developed independently, follow the same trend—symbol dependencies are better than include dependencies—which mitigates some of the potential bias. Furthermore, actual developers or architects of the projects aided in the production of our ground-truth architectures, further reducing any bias introduced in those architectures.

5.6.4 Number of Projects Studied

We have evaluated recovery techniques on only five systems, which limits our study's generalizability to other systems. Adding more projects is challenging. First, manually recovering the ground-truth architecture of a test project is time-consuming and requires the help of an expert with deep knowledge of the project [65]. Second, the projects studied need to be compatible with the tools used to extract dependencies. For example, the C++ projects evaluated need to be compilable with Clang. To mitigate this threat, we selected systems of different sizes, functionalities, architecture paradigms, and languages.

5.7 Summary

This work evaluates the impact of using more accurate symbol dependencies, versus the less accurate include dependencies used in previous studies, on the accuracy of automatic architecture recovery techniques. We also study the effect of different factors on the accuracy and scalability of recovery techniques, such as the type of dynamic bindings resolution, the granularity-level of the dependencies and whether the dependencies are direct or transitive. We studied nine variants of six architecture recovery techniques on five open-source systems. To perform our evaluation, we recovered the ground-truth architecture of Chromium, and updated ArchStudio and Bash architectures. In addition, we proposed a simple but novel submodule-based architecture recovery technique to recover preliminary versions of ground-truth architectures. In general, each recovery technique extracted a better quality architecture when using symbol dependencies instead of the less-detailed include dependencies. Working with direct dependencies at module level also helps with obtaining a more accurate recovered architecture. Finally, it is important to carefully choose

the type of dynamic-bindings resolution when working with symbol dependencies, as it can have a significant impact on the quality of the recovered architectures.

In some sense this general conclusion that quality of input affects quality of output is not surprising: the principle has been known since the beginning of computer science. Butler et al. [28] attribute it to Charles Babbage, and note that the acronym "GIGO" was popularized by George Fuechsel in the 1950's. What is surprising is that this issue has not previously been explored in greater depth in the context of architecture recovery. Our results show that not only does each recovery technique produce better output with better input, but also that the highest scoring technique often changes when the input changes.

There are other dimensions of architecture recovery that are worthy of future exploration, such as: recovering nested architectures; evaluating the usefulness of the recovered architecture to do specific maintenance tasks; and resolving function pointers and dynamic bindings.

The results presented here clearly demonstrate that there is room for more research both on architecture recovery techniques and on metrics for evaluating them.

Availability The ground-truth architectures are available at http://asset.uwaterloo.ca/ArchRecovery.

Chapter 6

Future Work

The projects in this thesis could be further supported and expanded with the following future work. Section 6.1 details future work on NMT-based program repair, Section 6.2 lists some possible extensions on finding bugs in electronic documents and readers, Section 6.3 proposes improvement in software architecture recovery. Finally, Section 6.4 proposes applications of machine learning and deep learning on other software engineering tasks.

6.1 Automatic Program Repair: Better Abstraction for Scalable NMT-based program repair

While CoCoNuT's results from Chapter 3 are promising, there are still room for improvement.

First, CoCoNuT and other NMT-based approaches use a very large vocabulary. NMT is effective for NLP tasks because most natural languages contain about 50,000 to 80,000 words. However, source code can have an infinite number of tokens. For example, CoCoNuT contains a vocabulary of 139,423 tokens, making it challenging (and expansive) for the network to learn and generalize.

Second, despite such large vocabulary sizes, there are a lot of rare tokens that cannot be captured, also called out-of-vocabulary (oov) tokens, even when the vocabulary contains several hundreds of thousand tokens (e.g., project specific tokens that only appear in specific test benchmarks). It is very challenging for NMT-based techniques like CoCoNuT to fix bugs that include oov tokens because the correct patch is outside of the search space.

```
- steps.add(new Pair<Integer,Integer>(start, helper));
+ steps.add(new Pair<Integer,Integer>(start, end));
```

(a) Patch for HANOI in `QuixBugs` for Java.

```
- v1.v2(new v3<Integer,Integer>(v4, v5));
+ v1.v2(new v3<Integer,Integer>(v4, v6));
```

(b) Abstracted version of the buggy and clean lines

```
v1={steps}, v2={add}, v3={Pair}, v4={start}
v6={height, end}
```

(c) Mapping between abstracted variables and identifiers in the source code

Figure 6.1: Bug, abstraction, and mapping for the HANOI bug in `QuixBugs` that CoCoNuT cannot fix without using abstracted template.

A future direction to expand CoCoNuT could address the vocabulary size issue by abstracting all the tokens to the form: v concatenated with a unique number. Instead of learning the correct patch, which is too difficult without knowing the entire project, the NMT model focuses on learning the correct template to transform a buggy line to a fixed line (e.g., insert a new variable or add a function call) [251].

Figure 6.1 shows an example of such an abstraction. Figure 6.1a shows the buggy and clean lines, as starting with a '-' and '+' respectively. The buggy line is first abstracted (line started with a '-' in Figure 6.1b) and the model uses historical data to generate an abstracted patch (line with a '+' in Figure 6.1b). Figure 6.1c shows the constraints automatically extracted from the source code we use to build a complete patch. $v1$ to $v4$ are inferred from the buggy line while $v6$ is inferred by automatically extracting variables in scope that match the signature of `Pair<Integer, Integer>` (i.e., variables of type `Integer`). There are two such variables in scope, but only the patch using $v6$=`end` passes all test cases. Previous work that does not use abstraction fails to generate the correct patch because it does not know which variables are in scope and attempts to replace `helper` with variables from other projects that appear in the training data.

Such an approach significantly reduces the size of the vocabulary used by CoCoNuT, from 139,423 to 2,651. As a result, it is faster to train, contains more correct patches in its search space, and overall fixes more bugs than previous work. Tufano et al. [238] and Wei [251] use a similar abstraction but did not solve the challenges created by such an abstraction detailed below.

Even if the model can generate the correct abstracted fix, it might not be ranked first. Furthermore, for each abstracted fix in a realistic project there are potentially hundreds or even thousands of possible variables to choose from. For example, the bug Closure 92 in

`Defects4J` is fixed by replacing the method `indexOf` with the method `lastIndexOf`. Such method are from an external library that are imported using wild card import statements. As a result, in practice there are hundreds of methods in scope for this patch. Therefore, even if the correct abstracted patch has a high rank, a naive approach to reconstruct the correct patch might still generate thousands of patches before reaching the correct one, which is unscalable. Even the one-file programs from QuixBugs can have up to 200 potential callable functions in scope from common libraries.

This issue can be addressed in future work by **extracting software engineering knowledge.** Specifically, we can keep track of variables in scope of the buggy line, extract functions' signature (i.e., number of parameters, and return type) and perform type analysis to narrow down the number of candidate tokens.

While such information helps narrow down the set of possible variables, there are still many candidate variables or functions to reconstruct a patch that need to be ranked efficiently.

Preliminary evaluation using this approach fixed 17 bugs in the Java `QuixBugs` [135] benchmark while CoCoNuT, the current state-of-the-art on this benchmark presented in Chapter 3, only fixed 13 bugs.

6.2 Detecting Bugs in PDF Documents and Readers

In the future, we plan to automatically fix the detected bugs in PDF files to ensure consistency across readers. The fault localisation component of our technique based on delta-debugging makes it possible to narrow down the problem to specific objects in the document. Then we could focus on removing or fixing those problematic objects. Fixing PDF documents is not a trivial task due to document format constraints such as cross-reference table that lists object offsets in the file, or object numbers which are used to cross-reference objects within the document. Even removal of a problematic object may result in a need to update other objects referencing the removed one or object offsets in the mentioned cross-reference table.

Another potential line of work is to extend the technique to other operating systems and document types. That would involve preparing the corresponding virtual machines and tuning our screenshot capture tools. Other document/file types might also need a different similarity metric—a study of ROC curves similar to the presented one might be necessary to pick the best metric for the task.

Finally, with more engineering effort we could improve system's performance and try to analyze randomly chosen pages of the document or a whole document, rather than the first page only. We believe that the current performance is reasonable given the data set size. However, more work in this area would make it possible to further scale up the technique.

6.3 Software Architecture Recovery

Dependencies. The work described in Chapter 5 explores whether the type of dependencies used affects the quality of the architecture recovered, and answers in the affirmative: Each recovery technique improves if more detailed input dependencies are used.

The results of Chapter 5 show, however, that any attempted evaluation of architecture recovery techniques must be careful about dependencies: For example, if we look at the best architecture recovery technique to recover Bash, MoJoFM would select a different best technique in four out of five cases with different input dependencies; $c2c_{cvg}$ in 3/5 cases; and normalized TurboMQ in 2/5 cases. a2a is more stable and would select Bunch-SAHC in all the cases, but a2a also shows that most of the techniques perform similarly for Bash when using similar dependencies. If we look at the other projects, we also observe that none of the metrics always pick the same best recovery technique when using different dependencies.

In Chapter 5, we evaluate architecture recovery techniques using source-code dependencies. Other types of dependencies can alternatively be used. For example, one can look at a developers' activity (e.g., files modified together) to obtain code dependencies [113] and further work is necessary to evaluate if completely different types of dependencies such as directory structure, historical information or developer's activity can be use in the context of automatic architecture recovery.

In addition, we do not consider weighting dependencies. For example, consider FileA that uses one symbol from FileB, and FileC that uses 20 symbols from FileB. Intuitively, it seems that FileB and FileC are more connected than FileA and FileB. Unfortunately, the current implementations of the architecture recovery techniques do not consider weighted graphs. Using weighted dependencies could also be a way to improve the quality of the recovered architectures.

Nested architectures. The architecture recovery techniques evaluated in Chapter 5 all recover "flat", i.e., non-hierarchical architectures. We focus on flat architectures for several reasons. First, for 4/5 systems we only have access to a flat ground-truth architecture. Second, the existing automatic architecture recovery techniques we evaluate only recover flat architectures.

In previous work on obtaining ground-truth architectures [65], results indicate that architects do not necessarily agree as to the importance of having a nested or flat architecture. However, when discussing with Google developers during the recovery of Chromium's ground truth, it appeared that they view their architecture as a nested architecture in which files are clustered into small entities, themselves clustered into larger entities. Some work has been done on improving metrics to compare nested architectures [221, 222], but little work has been done on proposing and evaluating automatic techniques for recovering nested architectures. A proper treatment of nested architectures, while out of scope of this work, is an important area for future research.

Multiple Ground-Truth Architectures. The work in Chapter 5 relies on several metrics used for evaluation of architecture recovery, some of which require a ground-truth architecture that might not be unique. More empirical work is needed to explore the idea of multiple ground-truth architectures for a given system. One possible direction is to conduct ground-truth extraction with different groups of engineers on the same system. Another direction would be to have system engineers develop ground-truth architectures starting from automatically recovered architectures. Ground-truth architectures are important for quality architecture-recovery evaluation and deserve further examination.

6.4 Explainable Defect Prediction:

Much work has been done using machine learning for defect prediction [5, 62, 72, 72, 73, 88, 96, 97, 98, 102, 133, 171, 172, 173, 174, 178, 181, 188, 216, 232, 248, 264, 266, 272, 275]. However one of the main issue to address for defect prediction to be widely adopted by industry is to improve the **explainability of the results** [96, 97, 129].

For example, the empirical study led by Jiarpakdee et al. [97] found that 55% of the users found model agnostic explanations necessary to understand defect prediction results and make them actionable. However, the effort to provide good explanations to the results have been limited, varying to presenting explanations based on features used in the model [96] or simply providing similar changes that occurred in the training set [97].

One possible application of deep learning for defect prediction could be to automatically generate explanation for the detected bugs using similar approaches that have been used to automatically generate commit messages [95, 144] or comments from source code [86].

Chapter 7

Conclusion

This thesis demonstrates that machine learning can be used effectively for many tasks related to software dependability. We illustrated this fact with three projects in three different domains of software dependability.

In Chapter 3, we propose a new NMT architecture, CoCoNuT, and demonstrate its efficacity for automatic program repair, fixing 493 bugs on six benchmarks in four different programming languages.

In Chapter 4, we first introduce the novel problem of cross-reader inconsistencies, which are caused by bugs in readers and files and shows that cross-reader inconsistencies are common. Furthermore, we propose a ML-based approach to select bug-revealing inputs and combined it to a new technique that detect and localize inconsistencies automatically. Our approach has detected 30 unique bugs on Linux and Windows systems. We also reported 33 bugs to developers, 17 of which have already been confirmed or fixed.

In chapter 5, we evaluate the impact of using more accurate symbol dependencies on the accuracy of automatic architecture recovery techniques. We also study the effect of different factors on the accuracy and scalability of recovery techniques, such as the type of dynamic bindings resolution, the granularity-level of the dependencies and whether the dependencies are direct or transitive. We studied nine variants of six architecture recovery techniques on five open-source systems.

Together these projects show that, applied correctly, machine learning and deep learning can significantly improve software dependability.

References

[1] Understand, Scitools.com. https://scitools.com.

[2] Fairseq-py. https://github.com/pytorch/fairseq, 2018.

[3] Archlinux, [wontfix] wrong colours in adobe reader (acroread). https://bbs.archlinux.org/viewtopic.php?id=193918.

[4] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. Sos-repair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering*, 2019.

[5] Igor Aizenberg and Claudio Moraga. Multilayer feedforward neural network based on multi-valued neurons (mlmvn) and a backpropagation learning algorithm. *Soft Computing*, 11(2):169–183, 2007.

[6] Mahdi Nasrullah Al-Ameen, Md Monjurul Hasan, and Asheq Hamid. Making findbugs more powerful. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pages 705–708. IEEE, 2011.

[7] Carol V Alexandru. Guided code synthesis using deep neural networks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1068–1070. ACM, 2016.

[8] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[10] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[12] Periklis Andritsos, Panayiotis Tsaparas, Renée J Miller, and Kenneth C Sevcik. LIMBO: Scalable Clustering of Categorical Data. In *Adv. Database Technol. - EDBT 2004*, pages 531–532. 2004.

[13] Periklis Andritsos and Vassilios Tzerpos. Information-Theoretic Software Clustering. *IEEE Trans. on Softw. Eng.*, 31(2), February 2005.

[14] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[15] Mahir Arzoky, Stephen Swift, Allan Tucker, and James Cain. Munch: An Efficient Modularisation Strategy to Assess the Degree of Refactoring on Sequential Source Code Checkings. In *Proc. ICST Workshops*, pages 422–429, 2011.

[16] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 328–332. IEEE, 2019.

[17] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[18] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341, 1996.

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[20] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[21] Dirk Beyer. Relational Programming with CrocoPat. In *Proc. ICSE*, pages 807–810, Shanghai, China, 2006. IEEE.

[22] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[23] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux As a Case Study: Its Extracted Software Architecture. In *Proc. ICSE*, pages 555–563, New York, NY, USA, 1999. ACM.

[24] Stefan Brahler. Analysis of the android architecture. *Karlsruhe institute for technology*, 7, 2010.

[25] A. Brown and G. Wilson. In *The Architecture of Open Source Applications*. Lulu, 2011.

[26] Bugzilla, bug 94260 - pdf file doesn't load or is displayed inconsistently. https://bugs.freedesktop.org/show_bug.cgi?id=94260.

[27] Bugzilla@Mozilla, Bug 1244729 - [PDF Viewer] Incorrect PDF display (large portions of the map appear as black) . https://bugzilla.mozilla.org/show_bug.cgi?id=1244729.

[28] Jill Butler, William Lidwell, and Kritina Holden. *Universal Principles of Design*. Rockport Publishers, Gloucester, MA, 2nd edition, 2010.

[29] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Tree2tree neural translation model for learning source code changes. *arXiv preprint arXiv:1810.00314*, 2018.

[30] Chris Chedgey, Paul Hickey, Paul O'Reilly, and Ross McNamara. *Structure101*.

[31] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning Affordance for Direct Perception in Autonomous Driving. In *ICCV*, 2015.

[32] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 637–647. IEEE, 2017.

[33] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[34] Shamil Chollampatt and Hwee Tou Ng. A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction. 2018.

[35] Shamil Chollampatt and Hwee Tou Ng. A multilayer convolutional encoder-decoder neural network for grammatical error correction. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, February 2018.

[36] Shauvik Roy Choudhary. Detecting cross-browser issues in web applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1146–1148. IEEE, 2011.

[37] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 171–180. IEEE, 2012.

[38] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[39] Chromium Bug Tracker, PDF's not displaying with Chromes PDF Distiller. https://code.google.com/p/chromium/issues/detail?id=333918.

[40] Google Chrome Help Forum, PDF viewer bug with tcpdf. https://productforums.google.com/forum/#!msg/chrome/tVNKJhiv-XQ/tH9RZyPlJGwJ.

[41] Aviad Cohen, Nir Nissim, and Yuval Elovici. Maljpeg: Machine learning based solution for the detection of malicious jpeg images. *IEEE Access*, 8:19997–20011, 2020.

[42] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *Proc. CSMR*, pages 35–44. IEEE, 2011.

[43] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 47–57. ACM, 2014.

[44] Complex-wavelet structural similarity index (cw-ssim). http://www.mathworks.com/matlabcentral/fileexchange/43017-complex-wavelet-structural-similarity-index--cw-ssim-.

[45] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 933–941. JMLR.org, 2017.

[46] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101. Springer, 1995.

[47] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th international conference on Software engineering*, pages 176–185. ACM, 2005.

[48] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2019.

[49] Lei Ding and Nenad Medvidovic. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. In *Proc. WICSA*, pages 191–200, Washington, DC, USA, 2001. IEEE CS Press.

[50] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. Patching as translation: the data and the metaphor. In *ASE'20*, 2020.

[51] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 85–91. ACM, 2016.

[52] Cyntrica Eaton and Atif M Memon. An empirical approach to evaluating web application compliance across diverse client platform configurations. *International Journal of Web Engineering and Technology*, 3(3):227–253, 2007.

[53] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. Caradoc: a pragmatic approach to pdf parsing and validation. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 126–139. Ieee, 2016.

[54] IEEE CONTENT ENGINEERING. *PDF Specification for IEEE Xplore (Part A-Core Requirements)*. IEEE, 2008.

[55] Jean-Marie Favre. Cacophony: Metamodel-driven software architecture reconstruction. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 204–213. IEEE, 2004.

[56] Jean-Marie Favre, Frédéric Duclos, Jacky Estublier, Remy Sanlaville, and Jean-Jacques Auffre. Reverse engineering a large component-based software product. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 95–104. IEEE, 2001.

[57] Mattia Fazzini and Alessandro Orso. Automated cross-platform inconsistency detection for mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 308–318. IEEE, 2017.

[58] Patrick J. Finnigan, Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi A Muller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong. The software bookshelf. *IBM systems Journal*, 36(4):564–593, 1997.

[59] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A Cliché-Based Environment to Support Architectural Reverse Engineering. In *Proc. ICSM*, pages 319–328. IEEE CS Press, 1996.

[60] Roberto Fiutem, Giulio Antoniol, Paolo Tonella, and Ettore Merlo. ART: an Architectural Reverse Engineering Environment. *Journal of Software Maintenance*, 11(5):339–364, 1999.

[61] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[62] Wei Fu and Tim Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 72–83, 2017.

[63] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[64] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *Proc. ASE*, pages 486–496. IEEE, 2013.

[65] Joshua Garcia, Ivo Krka, Chris Mattmann, and Nenad Medvidovic. Obtaining Ground-truth Software Architectures. In *Proc. ICSE*, ICSE '13, pages 901–910, Piscataway, NJ, USA, 2013. IEEE Press.

[66] Joshua Garcia, Duc Le, Daniel Link, Arman Shahbazian Pooyan Behnamghader, Eder Figueroa Ortiz, and Nenad Medvidovic. An empirical study of architectural

change and decay in open-source software systems. Technical report, USC-CSSE, 2014.

[67] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. Enhancing Architectural Recovery Using Concerns. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 552–555, 2011.

[68] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *digital investigation*, 6:S2–S11, 2009.

[69] Tao Ge, Furu Wei, and Ming Zhou. Fluency Boost Learning and Inference for Neural Grammatical Error Correction. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 1:1–11, 2018.

[70] Tao Ge, Furu Wei, and Ming Zhou. Reaching Human-level Performance in Automatic Grammatical Error Correction: An Empirical Study. (3):1–15, 2018.

[71] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. pages 1243–1252, 2017.

[72] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 789–800. IEEE, 2015.

[73] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 171–180. IEEE, 2012.

[74] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21, 2012.

[75] Cinthya Grajeda, Frank Breitinger, and Ibrahim Baggili. Availability of datasets for digital forensics–and what is missing. *Digital Investigation*, 22:S94–S105, 2017.

[76] Alan Grosskurth and Michael W. Godfrey. A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser, 2007.

[77] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944. ACM, 2018.

[78] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *arXiv preprint arXiv:2007.08095*, 2020.

[79] Rahul Gupta. *Deep Learning for Bug Localization and Program Repair*. PhD thesis, 2020.

[80] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351, 2017.

[81] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: A benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.

[82] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156. ACM, 2016.

[83] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*, pages 78–88. IEEE, 2009.

[84] Ahmed E Hassan, Zhen Ming Jiang, and Richard C Holt. Source versus object code extraction for recovering software architecture. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[85] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE*, pages 763–773, 2017.

[86] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217, 2020.

[87] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, pages 888–891. ACM, 2018.

[88] Qiao Huang, Xin Xia, and David Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170. IEEE, 2017.

[89] Quan Huynh-Thu and Mohammed Ghanbari. Scope of validity of psnr in image/video quality assessment. *Electronics letters*, 44(13):800–801, 2008.

[90] ISO. *Part 1, Graphic Technology: Pre Press Digital Data Exchange*. ISO, 2001.

[91] ISO. *Document Management: Electronic Document File Format for Long-term Preservation*. ISO, 2005.

[92] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity – a measure of the difficulty of speech recognition tasks. *Journal of the Acoustical Society of America*, 62:S63, November 1977. Supplement 1.

[93] Young-Seob Jeong, Jiyoung Woo, and Ah Reum Kang. Malware detection on byte streams of pdf files using convolutional neural networks. *Security and Communication Networks*, 2019, 2019.

[94] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. pages 298–309, 2018.

[95] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. IEEE Press, 2017.

[96] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee, 2013.

[97] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 2020.

[98] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 414–423, 2014.

[99] Mona Erfani Joorabchi, Mohamed Ali, and Ali Mesbah. Detecting inconsistencies in multi-platform mobile apps. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 450–460. IEEE, 2015.

[100] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Shubha Guha, and Kenneth Heafield. Approaching Neural Grammatical Error Correction as a Low-Resource Machine Translation Task. (2016):595–606, 2018.

[101] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[102] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[103] Masahiro Kaneko, Yuya Sakaizawa, and Mamoru Komachi. Grammatical Error Detection Using Error- and Grammaticality-Specific Word Embeddings. *Proceedings ofthe The 8th International Joint Conference on Natural Language Processing*, (2016):40–48, 2017.

[104] Ah Reum Kang, Young-Seob Jeong, Se Lyeong Kim, Jonghyun Kim, Jiyoung Woo, and Sunoh Choi. Detection of malicious pdf based on document structure features and stream objects. *Journal of The Korea Society of Computer and Information*, 23(11):85–93, 2018.

[105] Ah Reum Kang, Young-Seob Jeong, Se Lyeong Kim, and Jiyoung Woo. Malicious pdf detection model against adversarial attack built from benign pdf containing javascript. *Applied Sciences*, 9(22):4764, 2019.

[106] Rick Kazman and S Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

[107] Holger M Kienle and Hausi A Müller. Rigi—an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247–263, 2010.

[108] Jindae Kim and Sunghun Kim. Automatic patch generation with context-based change application. *Empirical Software Engineering*, 24(6):4071–4106, 2019.

[109] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 481–490. IEEE, 2011.

[110] MT Kirsch, VA Regenie, ML Aguilar, O Gonzalez, M Bay, ML Davis, CH Null, RC Scully, and RA Kichak. Technical support to the national highway traffic safety administration (nhtsa) on the reported toyota motor corporation (tmc) unintended acceleration (ua) investigation. *NASA Engineering and Safety Center Technical Assessment Report (January 2011)*, 2011.

[111] Kenichi Kobayashi, Manabu Kamimura, Koki Kato, Keisuke Yano, and Akihiko Matsuo. Feature-gathering Dependency-based Software Clustering using Dedication and Modularity. *Proc. ICSM*, 0:462–471, 2012.

[112] Jeffrey Koch and Kendra Cooper. Aovis: A model-driven multiple-graph approach to program fact extraction for aspectj/java source code. *Software Engineering: An International Journal*, 1, 2011.

[113] Martin Konôpka and Mária Bieliková. Software developer activity as a source for identifying hidden source code dependencies. In *SOFSEM 2015: Theory and Practice of Computer Science*, pages 449–462. Springer, 2015.

[114] Phil Koopman. A case study of toyota unintended acceleration and software safety. *Presentation. Sept*, 2014.

[115] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791*, 2018.

[116] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325, 2019.

[117] Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. Docovery: Toward generic automatic document recovery. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 563–574. ACM, 2014.

[118] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in pdf readers and files. *Empirical Software Engineering*, 23(6):3187–3220, 2018.

[119] Petri K Laine. The role of sw architecture in solving fundamental problems in object-oriented development of large embedded sw systems. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 14–23. IEEE, 2001.

[120] Pavel Laskov. Detection of malicious pdf files based on hierarchical document structure. In *In Proceedings of the Network and Distributed System Security Symposium, NDSS 2013*. The Internet Society, 2013.

[121] Pavel Laskov and Nedim Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 373–382. ACM, 2011.

[122] Duc Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An Empirical Study of Architectural Change in Open-Source Software Systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015)*, 2015.

[123] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An empirical study of architectural change in open-source software systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 235–245. IEEE Press, 2015.

[124] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

[125] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. 1:213–224, 2016.

[126] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[127] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2012.

[128] Timothy Lethbridge and Nicolas Anquetil. Comparative Study of Clustering Algorithms and Abstract Representations for Software Remodularization. *IEE Proceedings - Software*, 150(3):185–201, 2003.

[129] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead. Does bug prediction support human developers? findings from a google case study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 372–381. IEEE, 2013.

[130] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 318–328. IEEE, 2017.

[131] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 249–260. IEEE, 2017.

[132] Min Li, Yunzheng Liu, Min Yu, Gang Li, Yongjian Wang, and Chao Liu. Fepdf: a robust feature extractor for malicious pdf detection. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 218–224. IEEE, 2017.

[133] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.

[134] Yi Li, Wang Shaohua, and Tien N. Nguyen. DLfix: Context-based code transformation learning for automated program repair. In *Software Engineering (ICSE), 2020 IEEE/ACM 42nd International Conference on*. IEEE, 2020.

[135] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56. ACM, 2017.

[136] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[137] Daniel Link, Pooyan Behnamghader, Ramin Moazeni, and Barry Boehm. Recover and relax: Concern-oriented software architecture recovery for systems development and maintenance. In *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pages 64–73. IEEE, 2019.

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

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

[140] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 31–42, New York, NY, USA, 2019. ACM.

[141] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662. IEEE, 2018.

[142] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé François D Assise Bissyande, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020.

[143] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129. IEEE, 2018.

[144] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are

we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 373–384, 2018.

[145] Zhuo Ran Liu and Yang Liu. Exploiting Unlabeled Data for Neural Grammatical Error Detection. *Journal of Computer Science and Technology*, 32(4):758–767, 2017.

[146] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305, 2005.

[147] 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*, volume 2017, 2017.

[148] Fan Long et al. *Automatic patch generation via learning from successful human patches*. PhD thesis, Massachusetts Institute of Technology, 2018.

[149] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 80–90. IEEE, 2012.

[150] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[151] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 702–713. IEEE, 2016.

[152] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. Assessing test case prioritization on real faults and mutants. In *2018 IEEE international conference on software maintenance and evolution (ICSME)*, pages 240–251. IEEE, 2018.

[153] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[154] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 69–78. IEEE, 2015.

[155] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. *Proc. ICSE (SEIP)*, 2015.

[156] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidović, and Robert Kroeger. Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Transactions on Software Engineering*, 44(2):159–181, 2017.

[157] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–114, 2020.

[158] J MacQueen. On convergence of k-means and partitions with minimum average variance. In *Annals of Mathematical Statistics*, volume 36, page 1084. INST MATHEMATICAL STATISTICS IMS BUSINESS OFFICE-SUITE 7, 3401 INVESTMENT BLVD, HAYWARD, CA 94545, 1965.

[159] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[160] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.

[161] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 119–130. ACM, 2013.

[162] Ali Safari Mamaghani and Mohammad Reza Meybodi. Clustering of Software Systems Using New Hybrid Algorithms. In *Proc. CIT*, pages 20–25, 2009.

[163] S. Mancoridis, B. S. Mitchell, and C. Rorres. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Proc. IWPC*, pages 45–53, 1998.

[164] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proc. ICSM*, 1999.

[165] Onaiza Maqbool and Haroon Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Trans. Softw. Eng.*, 33(11):759–780, November 2007.

[166] Onaiza Maqbool and Haroon Atique Babri. The Weighted Combined Algorithm: A Linkage Algorithm for Software Clustering. In *Proc. CSMR*, pages 15–24. IEEE CS Press, 2004.

[167] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[168] WM McKeeman. Differential testing for software, digital tech. *J*, 10:100–107, 1998.

[169] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016.

[170] Nenad Medvidovic. ADLs and Dynamic Architecture Changes. In *Proc. ISAW*, ISAW '96, pages 24–27, New York, NY, USA, 1996. ACM.

[171] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–10, 2009.

[172] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 107–116. IEEE, 2010.

[173] Thilo Mende, Rainer Koschke, and Marek Leszak. Evaluating defect prediction models for a large evolving software system. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 247–250. IEEE, 2009.

[174] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, 2008.

[175] Ali Mesbah and Mukul R Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570. ACM, 2011.

[176] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925–936, 2019.

[177] Hrushikesh N. Mhaskar, Sergei V. Pereverzyev, and Maria D. van der Walt. A deep learning approach to diabetic blood glucose prediction. In *Front. Appl. Math. Stat.*, 2017.

[178] Bharavi Mishra, K Shukla, et al. Defect prediction for object oriented software using support vector based fuzzy classification model. *International Journal of Computer Applications*, 60(15), 2012.

[179] Brian S. Mitchell. A Heuristic Approach to Solving the Software Clustering Problem. In *Proc. ICSM*, pages 285–288. IEEE CS Press, 2003.

[180] Brian S. Mitchell and Spiros Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Trans. Softw. Eng.*, 32(3):193–208, March 2006.

[181] Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[182] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211*, 2015.

[183] Mozilla Support Forum, PDF.js not being displayed correctly. https://support.mozilla.org/en-US/questions/948061.

[184] Hausi Muller. Integrating information sources for visualizing java programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 250. IEEE Computer Society, 2001.

[185] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[186] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

[187] Gail C Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *ACM SIGSOFT Software Engineering Notes*, 20(4):18–28, 1995.

[188] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, 2017.

[189] Courtney Napoles and Chris Callison-Burch. Systematically Adapting Machine Translation for Grammatical Error Correction. *Proceedings of the 12th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 345–356, 2017.

[190] Liam O'Brien, Christoph Stoermer, and Chris Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, DTIC Document, 2002.

[191] Frolin S Ocariza, Jr, Karthik Pattabiraman, and Ali Mesbah. Vejovis: Suggesting fixes for javascript faults. In *Proceedings of the 36th International Conference on Software Engineering*, pages 837–847, 2014.

[192] Jugnu Gaur Ochin. Cross browser incompatibility: Reasons and solutions. *International Journal of Software Engineering & Applications (IJSEA)*, 2(3), 2011.

[193] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based Runtime Software Evolution. In *Proc. ICSE*, pages 177–186, Washington, DC, USA, 1998. IEEE CS Press.

[194] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch, 2017.

[195] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[196] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[197] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038. IEEE, 2019.

[198] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Problems and opportunities in training deep learning software systems: An analysis of variance. In *2020 IEEE/ACM Automated Software Engineering*. IEEE, 2020.

[199] matlabpyrtools. http://www.x.org/archive/X11R7.6/.

[200] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. Does genetic programming work well on automated program repair? In *2013 International Conference on Computational and Information Sciences*, pages 1875–1878. IEEE, 2013.

[201] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[202] Derek Rayside and Kostas Kontogiannis. Extracting Java Library Subsets for Deployment on Embedded Systems. *Sci. Comput. Program.*, 45(2-3):245–270, November 2002.

[203] Derek Rayside, Steve Reuss, Erik Hedges, and Kostas Kontogiannis. The Effect of Call Graph Construction Algorithms for Object-Oriented Programs on Automatic Clustering. In *Proc. IWPC*, pages 191–200. IEEE CS Press, 2000.

[204] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's δ for evaluating group differences on the NSSE and other surveys? In *annual meeting of the Florida Association of Institutional Research, February*, pages 1–3, 2006.

[205] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[206] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. X-pert: a web application testing tool for cross-browser inconsistency detection. In *Proceedings of*

the 2014 International Symposium on Software Testing and Analysis, pages 417–420. ACM, 2014.

[207] Tõnis Saar, Marlon Dumas, Marti Kaljuve, and Nataliia Semenenko. Cross-browser testing in browserbite. In *Web Engineering*, pages 503–506. Springer, 2014.

[208] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13, 2018.

[209] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. IEEE Press, 2017.

[210] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*, pages 13–24. IEEE Press, 2019.

[211] Keisuke Sakaguchi, Matt Post, and Benjamin Van Durme. Grammatical Error Correction with Neural Reinforcement Learning. 2017.

[212] Mehul P Sampat, Zhou Wang, Shalini Gupta, Alan Conrad Bovik, and Mia K Markey. Complex wavelet structural similarity: A new image similarity index. *Image Processing, IEEE Transactions on*, 18(11):2385–2401, 2009.

[213] Eddie A Santos, Joshua C Campbell, Abram Hindle, and José Nelson Amaral. Finding and correcting syntax errors using recurrent neural networks. *PeerJ PrePrints*, 2017.

[214] Samir G Sayed and Mohmed Shawkey. Data mining based strategy for detecting malicious pdf files. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 661–667. IEEE, 2018.

[215] Allen Schmaltz, Yoon Kim, Alexander M. Rush, and Stuart M. Shieber. Adapting Sequence Models for Sentence Correction. 2017.

[216] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 18–27, 2006.

[217] Scikit learn. http://scikit-learn.org/stable/.

[218] Hossain Shahriar, Komminist Weldemariam, Thibaud Lutellier, and Mohammad Zulkernine. A model-based detection of vulnerable and malicious browser extensions. In *2013 IEEE 7th International Conference on Software Security and Reliability*, pages 198–207. IEEE, 2013.

[219] Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine, and Thibaud Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 47:66–84, 2014.

[220] Flash Sheridan. Practical testing of a c99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[221] Mark Shtern and Vassilios Tzerpos. A framework for the comparison of nested software decompositions. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 284–292. IEEE, 2004.

[222] Mark Shtern and Vassilios Tzerpos. Lossless comparison of nested software decompositions. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 249–258. IEEE, 2007.

[223] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 239–248. ACM, 2012.

[224] Joshua A Solomon, Denis G Pelli, et al. The visual filter mediating letter identification. *Nature*, 369(6479):395–397, 1994.

[225] Hasan Sözer. Evaluating the effectiveness of multi-level greedy modularity clustering for software architecture recovery. In *European Conference on Software Architecture*, pages 71–87. Springer, 2019.

[226] Margaret-Anne D Storey, Kenny Wong, and Hausi A Müller. Rigi: a visualization environment for reverse engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 606–607. ACM, 1997.

[227] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. *Practical virtual method call resolution for Java*, volume 35. ACM, 2000.

[228] Nikita Synytskyy, Richard C Holt, and Ian Davis. Browsing software architectures with lsedit. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 176–178. IEEE, 2005.

[229] Damian A Tamburri and Rick Kazman. General methods for software architecture recovery: a potential approach and its evaluation. *Empirical Software Engineering*, 23(3):1457–1489, 2018.

[230] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[231] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 180–182. IEEE Press, 2017.

[232] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2018.

[233] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 19–20, 2020.

[234] Swizec Teller. *Data Visualization with D3.Js.* Packt Publishing, 2013.

[235] Paolo Tonella, Roberto Fiutem, Giuliano Antoniol, and Ettore Merlo. Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic -A Case Study. In *Proc. WCRE*, pages 198–207, 1996.

[236] Yuki Tsuchitoi and Hideki Sugiura. 10 mloc in your office copier. *IEEE software*, 28(6):93, 2011.

[237] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, pages 832–837, 2018.

[238] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[239] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, page 4. ACM, 2011.

[240] Vassilios Tzerpos and R. C. Holt. ACDC : An Algorithm for Comprehension-Driven Clustering. In *Proc. WCRE*, pages 258–267. IEEE, 2000.

[241] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. In *ICLR 2019*, 2019.

[242] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[243] F Waldman. Lattix LDM. In *8th International Design Structure Matrix Conference, Seattle, Washington, USA, October 24-26*. 2006.

[244] Jinyong Wang and Ce Zhang. Software reliability prediction using a deep learning model based on the rnn encoder–decoder. *Reliability Engineering & System Safety*, 2017.

[245] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[246] Pei Wang. Generating accurate dependencies for large software. 2013.

[247] Pei Wang, Jinqiu Yang, Lin Tan, Robert Kroeger, and David Morgenthaler. Generating Precise Dependencies For Large Software. In *Proc. MTD*, pages 47–50, May 2013.

[248] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[249] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 297–308. IEEE, 2016.

[250] Song Wang, Jaechang Nam, and Lin Tan. Qtep: quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 523–534, 2017.

[251] Moshi Wei. Abstraction mechanism on neural machine translation models for automated program repair. Master's thesis, University of Waterloo, 2019.

[252] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.

[253] Zhihua Wen and Vassilios Tzerpos. An Effectiveness Measure for Software Clustering Algorithms. In *Proc. IWPC*, pages 194–203, 2004.

[254] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[255] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.

[256] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Comparison of Clustering Algorithms in the Context of Software Evolution. In *Proc. ICSM*, pages 525–535, 2005.

[257] Chenchen Xiao and Vassilios Tzerpos. Software Clustering Based on Dynamic Dependencies. In *Proc. CSMR*, CSMR '05, pages 124–133, Washington, DC, USA, 2005. IEEE CS Press.

[258] Tong Xiao, Tian Xia, Yi Yang, Chang Huang, and Xiaogang Wang. Learning from massive noisy labeled data for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2691–2699, 2015.

[259] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670. IEEE Press, 2017.

[260] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.

[261] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[262] Xvfb, x window system version 11 release 7.6. http://www.x.org/archive/X11R7.6/.

[263] 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*, pages 831–841. ACM, 2017.

[264] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[265] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[266] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168, 2016.

[267] Helen Yannakoudakis, Marek Rei, Øistein E Andersen, and Zheng Yuan. Neural Sequence-Labelling Models for Grammatical Error Correction. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2795–2806, 2017.

[268] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. Random program generator for java jit compiler test system. In *Third International Conference on Quality Software, 2003. Proceedings.*, pages 20–23. IEEE, 2003.

[269] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 2018.

[270] Christoph Zauner. *Implementation and benchmarking of perceptual image hash functions*. na, 2010.

[271] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[272] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 309–320. IEEE, 2016.

[273] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455. IEEE, 2019.

[274] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Junhao Wen. Improving deep-learning-based fault localization with resampling. *Journal of Software: Evolution and Process*, page e2312, 2020.

[275] Shi Zhong, Taghi M Khoshgoftaar, and Naeem Seliya. Unsupervised learning for expert-based software quality estimation. In *HASE*, pages 149–155. Citeseer, 2004.