

Code Similarity and Clone Search in Large-Scale Source Code Data

Chaiyong Ragkhitwetsagul



A dissertation submitted in fulfillment
of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science
University College London

September 30, 2018

Declarations

I, Chaiyong Ragkhitwetsagul, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis. The papers presented here are original work undertaken between September 2014 and July 2018 at University College London. They have been published or submitted for publication as listed below along with a summary of my contributions to the papers.

1. C. Ragkhitwetsagul, J. Krinke, D. Clark, Similarity of source code in the presence of pervasive modifications, in *Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation (SCAM '16)*, 2016.

Contribution: Dr. Krinke designed and started the initial experiment which I later continued. I incorporated more tools, handled the tool executions, collected the results, analysed and discussed the results with Dr. Krinke. The three authors wrote the paper.

2. C. Ragkhitwetsagul, J. Krinke, D. Clark, A comparison of code similarity analysers, *Empirical Software Engineering (EMSE)*, 2017.

Contribution: This publication is an invited journal extension of the SCAM '16 paper to a special issue of Empirical Software Engineering. I substantially expanded the experiment by doubling the data set size, rerunning the first and second experimental scenarios, and introducing three additional experimental scenarios under the supervision of Dr. Krinke. The paper has been written by the three authors.

3. C. Ragkhitwetsagul and J. Krinke, Using compilation/decompilation to enhance clone detection, in *Proceedings of the 11th International Workshop on Software Clone (IWSC '17)*, 2017.

Contribution: I designed and performed the experiment and analysed the results with suggestions from Dr. Krinke. The paper was written by the two authors.

4. C. Ragkhitwetsagul and J. Krinke, Awareness and experience of developers to outdated and license-violating code on Stack Overflow : An online survey, *UCL Research Note*, 2017.

Contribution: I completed the two surveys and analysed the results under the supervision of Dr. Krinke. The research note is written by me and revised by Dr. Krinke.

5. C. Ragkhitwetsagul, M. Paixoa, J. Krinke, G. Bianco, R. Oliveto, Toxic code snippets on Stack Overflow, Submitted to *IEEE Transactions on Software Engineering (TSE)*, 2018 (after the first round of Revise and Resubmit as New).

Contribution: I lead the project and performed all the experiments and the surveys under the supervision of Dr. Krinke. G. Bianco, an exchange student from University of Molise under the supervision of Dr. Oliveto, helped on the data collection from Stack Overflow. My PhD colleague, M. Paixao, was involved in the manual clone classification. The paper was written by me and revised by Dr. Krinke and Dr. Oliveto.

6. C. Ragkhitwetsagul, J. Krinke, Siamese: Scalable and incremental code clone search via multiple code representations,

Submitted to *Empirical Software Engineering (EMSE)*, 2018.

Contribution: I performed the whole study under the supervision of Dr. Krinke. The paper was written by me and revised by Dr. Krinke.

7. R. White, J. Krinke, E. Barr, C. Ragkhitwetsagul, F. Sarro, M. Al-arfaj, Exploiting test-to-code traceability links for reuse, Submitted to *The 33rd International Conference on Automated Software Engineering (ASE '18)*.

Contribution: I provided and configured the Siamese clone search engine as a code similarity tool for the RELATEST test recommendation tool. I also involved in the manual validations of the recommended tests. The paper was written by R. White, Dr. Krinke, me, and Dr. Barr.

In addition, the list below includes the papers that I authored or co-authored and published or submitted for peer review during the programmed of my study. They are not featured in this thesis.

1. C. Ragkhitwetsagul, M. Paixao, M. Adham, S. Busari, J. Krinke, J. H. Drake, Searching for configurations in clone evaluation – A replication study, in *Proceedings of the 8th Symposium on Search-Based Software Engineering (SSBSE '16)*, 2016.
2. M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, Are developers aware of the architectural impact of their changes?, in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*, 2017.
3. M. Alonaizan, D. Han, J. Krinke, C. Ragkhitwetsagul, D. Schwartz-Narbonne, and B. Zhu, Recommending related code reviews, Submitted to *IEEE Transactions on Software Engineering (TSE)*, 2018 (Major Revisions).
4. C. Ragkhitwetsagul, J. Krinke, B. Marnette, A picture is worth a thousand words: Code clone detection based on image similarity, in *Proceedings of the 12th International Workshop on Software Clones (IWSC '18)*, 2018.

5. J. Wilkie, Z. Al Halabi, A. Karaoglu, J. Liao, G. Ndungu, C. Ragkhitwetsagul, M. Paixao, and J. Krinke, Who's this? Developer identification using IDE event data, in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*, 2018.



Chaiyong Ragkhitwetsagul

30 September 2018

Date

Abstract

Software development is tremendously benefited from the Internet by having online code corpora that enable instant sharing of source code and online developer’s guides and documentation. Nowadays, duplicated code (i.e., code clones) not only exists within or across software projects but also between online code repositories and websites. We call them “online code clones.” They can lead to license violations, bug propagation, and re-use of outdated code similar to classic code clones between software systems. Unfortunately, they are difficult to locate and fix since the search space in online code corpora is large and no longer confined to a local repository.

This thesis presents a combined study of code similarity and online code clones. We empirically show that many code snippets on Stack Overflow are cloned from open source projects. Several of them become outdated or violate their original license and are possibly harmful to reuse. To develop a solution for finding online code clones, we study various code similarity techniques to gain insights into their strengths and weaknesses. A framework, called OCD, for evaluating code similarity and clone search tools is introduced and used to compare 34 state-of-the-art techniques on pervasively modified code and boiler-plate code. We also found that clone detection techniques can be enhanced by compilation and decompilation.

Using the knowledge from the comparison of code similarity analysers, we create and evaluate Siamese, a scalable token-based clone search technique via multiple code representations. Our evaluation shows that Siamese scales to large-scale source code data of 365 million lines of code and offers high search precision and recall. Its clone search precision is comparable to seven state-of-the-art clone

detection tools on the OCD framework. Finally, we demonstrate the usefulness of Siamese by applying the tool to find online code clones, automatically analyse clone licenses, and recommend tests for reuse.

Impact Statement

The knowledge and methods of software analysis presented in this thesis are beneficial to software quality improvement and could have impacts on both software engineering research and industry.

The phenomenon of code duplication, i.e., code cloning, is well-known in both research and industry, although sometimes with different terms used to describe it. There are two camps of beliefs that clones lead to bug propagation and degrades software maintenance, and that clones rarely relate to bugs and do not affect software quality. However, they both agree that clones have to be made explicit. This thesis strengthens the body of knowledge in code clone research by studying a new type of code clones, a cloning activity to and from online sources such as Stack Overflow programming Q&A website.

The thesis shows that the result of such cloning for which we call *online code clones*, have at least two issues: outdated code and software license incompatibility. Since Stack Overflow is a popular website with 7.6 million users, the issues from online code cloning can affect a large number of programmers around the world. The findings lead to an urgent need to mitigate the issues, both by research and Stack Overflow itself.

On the research side, the thesis develops a scalable code clone search technique and a tool called Siamese, to provide a scalable solution to locate online code clones from large source code corpora. The thesis demonstrates that Siamese can be put to use to efficiently locate clones between Stack Overflow and a hundred open source projects. The technique can incorporate automated license analysis to detect violations of software licenses and has a potential to be transformed to a cloud

service that offers real-time checks for online code clones in any software projects.

On the industry side, the thesis calls for action from Stack Overflow to mitigate the issues of outdated code and software license violations. The survey results in the thesis clearly show that Stack Overflow users are aware of the issues and some of them need a guideline from Stack Overflow. Moreover, the website must collect the origin of the source code examples to check for their newer version and their original software license.

Lastly, the comparison of 34 code similarity analysers presented in the thesis is the largest in existence and potentially an invaluable guide for future users of similarity detection in source code. The findings can be used both in academia and outside academia on any studies or projects related to code similarity.

Acknowledgements

Many people have helped me during the four years of my PhD at UCL, London. I would not have made it this far without them. Thus, I would like to express my gratitude to the following people.

First and foremost, I would like to thank my supervisor, Dr. Jens Krinke, who made my life as a PhD student an enjoyable experience. I mostly appreciate Jens for his patience, guidance, and thoughtful comments on my work for these four years. Starting as a novice researcher who barely knows how to do research, I could see myself improving bit by bit in every meeting we had, and I am genuinely indebted for that.

I am also very grateful to my second supervisor, Dr. David Clark, for his friendly and insightful discussions.

I always feel fortunate to be a part of CREST, and it's mostly about the people. I have met several lecturers, RAs, and PhDs who eventually become friends and moral support. The people that I feel comfortable to share excitements or frustrations from my life and my study. In particular, I would like to thank Matheus Paixao, Carlos Gavidia, Bobby Bruce, and DongGyun Han for spending their time to hang out with me. My PhD would not be this fun without you guys.

I appreciate the friendship and support from all my Thai friends in London, especially the *Afternoon Tea* group including P'Kook Supanan Sucharit, Champ Chutipong Paraphantakul, Tohn Thanapong Intharah, and Bright Akkapon Wongkoblap, who started the PhD at roughly the same time and went through ups and downs together with me. I also thank Moch Akara Supratak, my friend and my future MUICT colleague, for being a great housemate and for many useful

suggestions on research.

I would like to express my acknowledgement to research collaborators that I worked with during the time of my PhD. Thank you Dr. Rocco Oliveto and Giuseppe Bianco for their help on the Stack Overflow work and thank you Robert White for incorporating Siamese in his work. Thank you to the people at Prodo.ai, especially Bruno Marnette and Sergio Giro, who gave me invaluable internship experience.

I would like to thank the Faculty of ICT and Mahidol University (i.e., my scholarship sponsors), who financially support me to fulfil my dream.

My deepest gratitude to my two special persons in London, who I consider a second family far away from home: Auntie Noy Vongdoen and Uncle Berm Watana Laomongkholchaisri. Thank you for the lovely accommodation and, more importantly, for embracing me as a member of your family. Your Thai dishes are always the best. They always helped me to get through my tough times.

I would like to thank my better half, Pinn Nichayada Ragkhitwetsagul, for all your love and encouragement. I really appreciate your understanding on my professional desire to pursue a PhD and your patience on waiting for me to finish it. I am very fortunate to have you in my life. Finally, I am deeply indebted to my wonderful parents and my lovely grandma, who have shaped me as myself today. As a kid from suburban Thailand, I could achieve this much in my life because of your endless love and full support in education. Thanks for always rooting for me and for your understanding in whatever I wanted to pursue in life. This PhD is for you.

Contents

1	Introduction	28
1.1	Problem Statements of the Thesis	31
1.2	Goal and Objectives	32
1.3	Contributions	33
1.4	Thesis Organisation	34
2	Literature Survey	36
2.1	The Spectrum of Program Similarity	36
2.2	Recent Publications in Code Similarity	37
2.3	Research Areas Involving Code Similarity	39
2.3.1	Code Clones	40
2.3.2	Software Plagiarism	47
2.3.3	Software License Compatibility	50
2.3.4	Other Applications of Code Similarity	51
2.4	Existing Code Similarity Detection Techniques and Tools	52
2.4.1	Metric-based Approaches	53
2.4.2	Text-based Approaches	54
2.4.3	Token-based Approaches	55
2.4.4	Tree-based Approaches	59
2.4.5	Graph-based Approaches	61
2.4.6	Compile Code-based Approaches	63
2.4.7	Model-based Approaches	65
2.4.8	Other Approaches	66

	<i>Contents</i>	13
2.5	Benchmarks for Comparing Code Similarity Tools	68
2.6	Scalable Code Similarity Measurement and Code Search Techniques	69
2.6.1	Scalable Clone Detection	69
2.6.2	Code Search	72
2.7	Chapter Summary	74
3	Awareness and Experience of Developers to Outdated and License-Violating Code on Stack Overflow: An Online Survey	75
3.1	Motivation	75
3.2	Stack Overflow: A Popular Programming Q&A Website	77
3.3	Terminology	78
3.4	Contributions	79
3.5	Research Methodology	79
3.5.1	Survey Objective	79
3.5.2	Survey Design and Schedule	80
3.5.3	Participant Selection	81
3.5.4	Data Analysis	82
3.6	Results and Discussions	83
3.6.1	The answerer survey	83
3.6.2	The visitor survey	92
3.6.3	Overall Discussion	99
3.7	Threats to Validity	101
3.7.1	Internal Validity	101
3.7.2	External Validity	101
3.8	Chapter Summary	102
4	An Empirical Study of Online Code Clones and Their Toxicity	103
4.1	Motivation	103
4.1.1	Online Code Clones	104
4.1.2	Toxic Code Snippets	105
4.2	Contributions	108

4.3	Empirical Study	109
4.3.1	Research Questions	109
4.3.2	Online Code Clone Detection	110
4.4	Results and Discussion	123
4.4.1	RQ1: Online Code Clones	123
4.4.2	RQ2: Patterns of Online Code Cloning	124
4.4.3	RQ3: Outdated Online Code Clones	129
4.4.4	RQ4: Software License Violations	135
4.4.5	Overall Discussion	138
4.5	Threats to Validity	142
4.5.1	Internal Validity:	142
4.5.2	External validity:	143
4.6	Related Work	143
4.7	Chapter Summary	144

5 OCD: A Framework for Evaluating Code Similarity and Clone Search

Tools		145
5.1	Motivation	146
5.2	Contributions	147
5.3	Background	149
5.3.1	Source Code Modifications	149
5.3.2	Obfuscation and Deobfuscation	151
5.3.3	Program Decompilation	153
5.4	The OCD Framework	154
5.4.1	The Similarity Report Template	155
5.4.2	Implementation of the Framework	157
5.4.3	Using the OCD Framework	161
5.5	Empirical Study	161
5.5.1	Research Questions	161
5.5.2	Code Similarity Detection Tools and Techniques	163
5.6	Experimental Scenarios	169

	<i>Contents</i>	15
5.6.1	Scenario 1 (Pervasive Modifications)	169
5.6.2	Scenario 2 (Reused Boiler-Plate Code)	174
5.6.3	Scenario 3 (Decompilation)	175
5.6.4	Scenario 4 (Ranked Results)	176
5.6.5	Scenario 5 (Pervasive Modifications + Boiler-plate Code) .	179
5.7	Results	181
5.7.1	RQ1: Performance Comparison	181
5.7.2	RQ2: Optimal Configurations	188
5.7.3	RQ3: Normalisation by Decompilation	191
5.7.4	RQ4: Reuse of Configurations	195
5.7.5	RQ5: Ranked Results	197
5.7.6	RQ6: Pervasive Modifications + Boiler-plate Code	206
5.7.7	Overall discussions	208
5.8	Threats to Validity	210
5.8.1	Construct validity	211
5.8.2	Internal Validity	211
5.8.3	External Validity	211
5.9	Related Work	212
5.10	Chapter Summary	214
6	Using Compilation/Decompilation to Enhance Code Clone Detection	216
6.1	Motivation	216
6.2	Contributions	217
6.3	Experimental Design	218
6.3.1	Experimental Framework	218
6.3.2	Software Systems	219
6.3.3	Tools	219
6.4	Results and Discussion	223
6.4.1	RQ1: Clone Agreement	223
6.4.2	RQ2: Decompilation Accuracy	226
6.4.3	RQ3: Characteristics of Disjoint Clones	227

	<i>Contents</i>	16
6.5	Overall Discussion	233
6.6	Threats to Validity	233
6.6.1	Internal Validity	233
6.6.2	External Validity	234
6.7	Related Work	234
6.8	Chapter Summary	234
7	Siamese: Scalable and Incremental Code Clone Search Engine	236
7.1	Motivation	236
7.2	Contributions	238
7.3	Siamese Clone Search Architecture	239
7.3.1	Indexing Phase	240
7.3.2	Querying Phase	240
7.3.3	Multi-Representation (MR)	241
7.3.4	Query Reduction (QR)	243
7.3.5	Scoring and Ranking of the Results	244
7.3.6	Incremental Updates	246
7.4	Siamese Implementation	247
7.4.1	Selection of N -gram Sizes	247
7.4.2	Choosing the Query Reduction Thresholds	248
7.5	Experimental Design	249
7.5.1	Data Sets	250
7.5.2	Error Measures	251
7.6	Evaluation and Results	252
7.6.1	RQ1: Multi-Representation and Query Reduction	253
7.6.2	RQ2: Search Accuracy	256
7.6.3	RQ3: Clone Ranking	265
7.6.4	RQ4: Scalability	269
7.6.5	RQ5: Incremental Updates	273
7.7	Threats to Validity	275
7.7.1	Internal and Construct Validity	275

<i>Contents</i>	17
7.7.2 External Validity	276
7.8 Chapter Summary	276
8 Applications of Siamese	277
8.1 Online Code Clone Detection on Stack Overflow	277
8.1.1 Similarity Threshold	278
8.1.2 Results	279
8.1.3 Discussion	282
8.2 Clone Search with Software License Analysis	282
8.2.1 Results	283
8.2.2 Discussion	286
8.3 Automating the Reuse of Tests	286
8.3.1 Searching for Similar Functions using Siamese	287
8.3.2 Results	287
8.3.3 Discussion	290
8.4 Chapter Summary	291
9 Conclusion and Future Work	292
9.1 Summary of Achievements	293
9.2 Summary of Future Work	294
Appendices	298
A Chapter 3: Answerer and Visitor Surveys	298
A.1 Open Comments from Stack Overflow Answerers	298
A.2 Stack Overflow Answerer Survey: Google Forms	304
A.3 Stack Overflow Visitor Survey: Google Forms	308
B Chapter 4: Outdated Code Snippets	312
C Chapter 5: The OCD Framework	314
C.1 The complete list of the optimal configurations	314
C.2 The OCD Framework User's Guide	315

<i>Contents</i>	18
D Chapter 7: Siamese – A User’s Guide	318
E Chapter 8: Detection of GitHub Projects’ Licenses	319
Bibliography	321

List of Figures

2.1	Spectrum of program similarity [Zhang et al., 2012]	37
2.2	Publications relating to code similarity in major software engineering venues in the past 10 years	38
2.3	Topics of code similarity publications	39
2.4	Type-1 clone pair with only differences in formatting	42
2.5	Type-2 clone pair with different data types, variables and formatting	42
2.6	Type-3 clone pair with modified/added/deleted/relocated statements	42
2.7	Type-4 clone pair with clearly distinct code structure but share the same semantic based on input/output. The clone fragment on the left implements bubble sort algorithm while the clone fragment on the right implements insertion sort.	43
3.1	Experience of the Stack Overflow answerers	83
3.2	Frequency of answering questions	84
3.3	Frequency of answering questions with code snippet(s)	84
3.4	Sources of code snippets in Stack Overflow answers	86
3.5	Percentage of answerers who are notified of outdated code in their Stack Overflow answers.	87
3.6	Frequency of being notified of outdated code in the answerers' answers.	87
3.7	Frequency of the answerers fixing their outdated code.	87
3.8	Awareness of answerers to Stack Overflow CC BY-SA 3.0 license .	89
3.9	Software license in Stack Overflow code snippets.	89

3.10 Frequency of the answerers checking license of their code snippets against Stack Overflow's CC BY-SA 3.0	90
3.11 Experience of the Stack Overflow visitors	93
3.12 Rankings of the resources developers use to solve programming problems	93
3.13 Frequency of copying code from Stack Overflow	94
3.14 Why did you copy and reuse code snippets from Stack Overflow? . .	95
3.15 Number of visitors who experienced a problem from reusing Stack Overflow code snippets and the frequency.	96
3.16 Problems from reusing Stack Overflow code snippets	96
3.17 Frequency of Stack Overflow visitors reporting the problems back to Stack Overflow discussion threads	97
3.18 Options that Stack Overflow visitors choose to report the problems from Stack Overflow snippets	97
3.19 Awareness of Stack Overflow visitor to CC BY-SA 3.0 license . . .	98
3.20 Attributions to Stack Overflow when reusing code snippets	98
3.21 Checking for the original license of Stack Overflow code snippets .	99
3.22 Checking for license conflicts from reusing Stack Overflow snippets	99
3.23 Legal issues found from reusing Stack Overflow code snippets . . .	99
4.1 An example of the two code fragments of <code>WritableComparator.java</code> . The one from the Stack Overflow post 22315734 (left) is outdated when compared to its latest version in the Hadoop code base (right). Its Apache v.2.0 license is also missing.	106
4.2 An example of the two code fragments of <code>StringUtils.java</code> . The one from the Stack Overflow post 801987 (left) is outdated when compared to its latest version in the Hadoop code base (right). The toxic code snippet is outdated code and has race conditions.	108
4.3 The Experimental framework	110
4.4 The result from clone merging using Bellon's ok-match criterion . .	118
4.5 Online code clone classification process	120

4.6	Age of QS online code clones.	126
4.7	The <code>findMethodToInvoke</code> that is found to be copied from Stack Overflow post 698283 to <code>POIUtils</code> class in <code>jstock</code>	126
4.8	Original sources of EX clone pairs	127
4.9	Outdated QS online clone pairs group by projects	129
5.1	Pervasive modifications found in a programming submission.	150
5.2	A boiler-plate code to create connection threads.	151
5.3	The OCD framework	154
5.4	An example similarity report	156
5.5	The same code fragments, a constructor of <code>MagicSquare</code> , after pervasive modifications, and compilation/decompilation.	171
5.6	Test data generation process	172
5.7	The graph shows the F-score and the threshold values of ncd-bzlib. The tool reaches the highest F-score when the threshold equals 37.	174
5.8	The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC).	182
5.9	The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC) for SOCO.	187
5.10	Trade off between precision and recall for 392 ccfx parameter settings. The default settings provide high precision but low recall against pervasive code modifications.	190
5.11	F-scores of 392 ccfx's b and t parameter values on pervasive code modifications	190
5.12	Comparison of tool performances (F1-score) before and after decompilation	193
5.13	Precision-at-n of the tools according to varied numbers of n against the OCD data set	199
5.14	Precision-at-n of the tools according to varied numbers of n against SOCO data set	200

5.15 Distribution of tools performance for each pervasive modification type	206
6.1 The experimental framework	219
6.2 The process of mapping decompiled clones to their original locations	221
6.3 Example of type-3 clones in <i>findDomainBounds()</i> and <i>findRange- Bounds()</i> that can be detected with type-2 configuration after de- compilation	230
6.4 Example of type-3 clones with different loops in <i>clearDomain- Markers()</i> and <i>clearRangeMarkers()</i>	231
6.5 The <i>clearDomainMarkers()</i> and <i>clearRangeMarkers()</i> that can be detected after decompilation	232
7.1 Siamese Architecture	240
7.2 An example code fragment of a binary search method	242
7.3 Java term rank and its document frequency	248
7.4 Search results with syntactic and semantic clones	264
7.5 A false positive clone pair containing methods inside a method	265
7.6 Another false positive clone pair containing similar code tokens	266
7.7 An example of Type-3/Type-4 cloned fragment returned as the 1st result	269
7.8 Indexing time (minutes)	272
7.9 Querying time (seconds)	272
7.10 Incremental index update time	275
8.1 A comparison of Siamese-NTR clone pairs to the previous results by Simian and SourcererCC (SM-SCC)	279
8.2 A contained type-3 clone pair reported by Siamese-NTR	281
8.3 The overview of RELATEST	286
8.4 An implementation of RELATEST with Siamese as a query processor.	288
C.1 An example of a tool running script (<code>compare_diff.sh</code>)	316

List of Tables

2.1	Code cloning patterns by Kapser and Godfrey [2008]	47
3.1	The Stack Overflow answerer taken the surveys	81
3.2	The Stack Overflow visitors taken the survey	83
4.1	Stack Overflow and Qualitas datasets	112
4.2	Configurations of Simian and SourcererCC	114
4.3	Number of online clones reported by Simian and SourcererCC . . .	116
4.4	The seven patterns of online code cloning	119
4.5	The definition of boiler-plate code	120
4.6	Investigated online clone pairs and corresponding snippets and Qualitas projects	123
4.7	Classifications of online clone pairs.	124
4.8	Qualitas projects associated with QS and UD online clone pairs . .	125
4.9	Six code modification types found when comparing the outdated clone pairs to their latest versions	130
4.10	Examples of the outdated QS online clones	132
4.11	Intents of code changes in 100 outdated code snippets	133
4.12	Clones of the 100 Stack Overflow outdated code snippets in 130,719 GitHub projects	133
4.13	License mapping of online clones (file-level)	136
4.14	Clones of the 214 Stack Overflow missing-license code snippets in 130,719 GitHub projects	138

5.1	List of pervasive code modifications offered by our source code and bytecode obfuscator, and compiler/decompilers	157
5.2	Tools with their similarity measures	167
5.3	Tools and parameters with chosen value ranges (DF denotes default parameters)	168
5.4	Descriptions of the 10 original Java classes in the generated OCD data set	169
5.5	Size of the data sets. The (generated) OCD data set in Scenario 1 has been compiled and decompiled before performing the detection in Scenario 2 ($\text{OCD}^{\text{decomp}}$). The SOCO data set is used in Scenario 3 and the SOCO with pervasive modification (SOCO^{ocd}) is used in Scenario 5.	170
5.6	10 pervasive code modification types	181
5.7	OC D data set (Scenario 1): rankings (R) by F-scores (F1) and optimal configuration of every tool and technique.	183
5.8	SOC O data set (Scenario 3): rankings (R) by F-scores (F1) and optimal configuration of every tool and technique.	186
5.9	ccfx's parameter settings for the highest precision and recall	189
5.10	Optimal configuration of every tool obtained from the generated ^{decomp} data set decompiled by Krakatau in Scenario 2 and their rankings (R) by F-scores (F1).	192
5.11	Wilcoxon signed-rank test of tools' performances before and after decompilation by Krakatau and Procyon ($\alpha = 0.05$).	193
5.12	Optimal configuration of every tool obtained from the generated ^{decomp} data set (decompiled by Procyon) in Scenario 2 and their rankings (R) by F-scores (F1).	196
5.13	Results after applying the best configurations (C_{gen}) from Scenario 1 to the SOCO data set and the derived best configurations for the SOCO set (C_{soco}).	197

5.14	Top-10 rankings of using prec@n, ARP, and MAP over the OCD data set with the tools' optimal configurations	198
5.15	Top-10 rankings of using prec@n, ARP, and MAP over the SOCO data set with the tools' optimal configurations	198
5.16	One-tailed randomization test with 100K samples of the ARP values from the OCD data set.	201
5.17	One-tailed randomization test with 100K samples of the MAP values from the OCD data set.	202
5.18	One-tailed randomization test with 100K samples of the ARP values from the SOCO data set.	202
5.19	One-tailed randomization test with 100K samples of the MAP values from SOCO data set.	204
5.20	F-scores of the tools on SOCO ^{ocd} using the default configurations (with optimised threshold). Highlighted values have F-score higher than 0.8.	205
6.1	Software systems	219
6.2	NiCad's configurations	220
6.3	Systems and clones found categorised by clone types. The numbers are clone pairs found only in a particular clone type (non-subsuming). C denotes “clone pairs” and Cf denotes “filtered clone pairs”.	225
6.4	Manual investigation results of clone pair candidates reported in $Cf_{orig-only}$ and $Cf_{decomp-only}$	227
6.5	Characteristics of disjoint clones reported in $Cf_{orig-only}$ and $Cf_{decomp-only}$	228
7.1	Representative tokens for specific token types	242
7.2	The four representations of the <code>binarySearch1</code> method generated from the MR module.	243
7.3	The data sets for Siamese evaluation	251

7.4	Search performance (MAP) with different combinations of code representations	255
7.5	Search performance improvement (MAP) after adding multi-representation and query reduction	255
7.6	Comparison of search performance (MAP) on the OCD data set (100 queries)	258
7.7	Comparison of search performance (MAP) on the SOCO data set (115 queries)	258
7.8	BigCloneBench Recall Measurements (All Clones)	261
7.9	BigCloneBench Recall Measurements (Intra-Project Clones)	261
7.10	BigCloneBench Recall Measurements (Inter-Project Clones)	261
7.11	BigCloneBench Precision Measurements	262
7.12	Type-3-only search: the boosting scores for the four code representations and the search accuracy	267
7.13	GitHub projects used for incremental update	274
8.1	Siamese execution on Stack Overflow and Qualitas corpus	278
8.2	The 413 SM-SCC online clone pairs that are found by Siamese	279
8.3	The data sets in the case study	282
8.4	GitHub projects with the highest no. of clones	284
8.5	License comparison of the clones	285
8.6	Ranked List Validation Methods.	289
8.7	Manual validation of recommendation lists (within-project)	289
8.8	Manual validation of recommendation lists (cross-project)	289
B.1	100 outdated cloned code snippets on Stack Overflow and their associated original projects	312
B.2	100 outdated cloned code snippets on Stack Overflow and their associated original projects (cont.)	313
C.1	The complete list of optimal configurations for Krakatau	314

E.1 A list of GitHub project's license keywords used by Siamese	320
---	-----

Chapter 1

Introduction

Code similarity is extensively exploited in software engineering research. One of the well-known applications is code clone detection, i.e., finding duplicated pieces of code in software. Code cloning occurs by programmers duplicating source code with or without further alterations. The reasons for cloning are varied, such as creating multiple versions of the software, reuse of a well-written code template, or adapting a functionality from existing code [Kapser and Godfrey, 2008]. Code cloning results in redundant pieces of code, which may lead to software maintenance issues [Juergens et al., 2009]. On the contrast, several studies have shown that clones are not always harmful and can possibly be beneficial in some situations [Kapser and Godfrey, 2006, Saini et al., 2016b]. In any case, clone researchers agree that code clones have to be made explicit to manage them properly. Thus, a large number of tools have been invented in both research and industry to detect clones. The latest clone survey by Rattan et al. [2013] reports 74 clone detection tools found in the literature.

Another area of code similarity research is software plagiarism detection. Software plagiarism is caused by code cloning with malicious intent to hide the ownership of the copied code [Duric and Gasevic, 2013]. It is a concern in both education and industry, and it can cause serious legal consequences. For example, Oracle declared enormous damage of nine billion dollars to the US Federal Circuit court claiming that Google has plagiarised their Java APIs in the Android operating system [Jeong, 2018].

Code similarity detection is also used to find software license violations. It has been reported in research that software license conflicts occur by code cloning [German et al., 2009, An et al., 2017]. Since each software project has a license which may differ from another, a careless code cloning from a strictly-licensed project to a permissively-licensed project will create a license violation. There are several cases in which such cloning produce serious legal issues [Gross, 2007, Lee, 2008, Economy, 2009]. A well-known case is when Linksys (now Cisco) violated the GPL license by reusing the Linux kernel and BusyBox in their closed-source WRT54G router [Hemel et al., 2011]. Software companies usually prevent this issue by performing code clone detection and license analysis between their software and open source software projects, a service offered by software auditing companies such as Black Duck Software.

With the rise of the Internet, several large-scale online code corpora such as GitHub (an online code repository and versioning system) or Stack Overflow (a popular programming Q&A website) containing possibly millions or billion lines of code, introduces several challenges to code similarity detection.

First, code can be cloned from *anywhere*. In the past, code cloning is confined to within or between software projects. Nowadays, online corpora such as GitHub and Stack Overflow have become rich sources of code examples. Since code can be freely and quickly accessed online, code cloning becomes easier than before. The issues from code cloning are also escalated by the ever-growing size of code bases available online. One might copy and reuse code snippets from GitHub projects without checking their original licenses, which may lead to legal issues. Sometimes a code fragment is put online without its original license, so the user of the code will never be aware of a potential violation. Also, code examples on Stack Overflow may be outdated and harmful for reuse because they are not tested or updated as frequently as in software projects.

Second, to detect clones in such large-scale source code data, a scalable code clone or code similarity tool is a necessity. The classical code clone and code plagiarism detection tools are not scalable enough to efficiently report similar code

artefacts in very large code corpora. For example, Tata Consultancy Services (TCS) faced difficulty when they tried to detect clones within their COBOL systems with many million lines of code [Sajnai, 2016]. The tools they used, CloneDR (a commercial tree-based clone detector) and Simian (a text-based clone detector), could not scale to such a large code base. The primary challenge of measuring similarity of source code on the Internet-scale is the quadratic execution time from n-to-n pairwise comparisons of code fragments or the expensive tree or graph comparisons in traditional code similarity detection approaches. Thus, we are seeking a scalable approach that reduces or avoids such comparisons while still accurately locate similar code fragments.

Third, although a few scalable clone detection tools have been recently proposed in the literature (e.g., Hummel et al. [2010], Sajnani et al. [2015, 2016], Svajlenko and Roy [2017], Saini et al. [2018]), the human effort to inspect pieces of code is yet limited [Miller, 1956]. As a result, a clone detector that reports a large number of clone pairs to a human investigator are not practically helpful in a case of licensing violation check, finding bug fixes, or plagiarism detection in programming submissions on large-scale source code data. The investigator needs a tool similar to the Google search engine that ranks the results by their relevance to a given code query so that she can investigate only a few top n cloned candidates. Code clone search engine, which receives a piece of code as a query and returns a ranked list of clones, is more suitable than clone detectors for these tasks. Moreover, most of the existing code clone detection tools do not support incremental updates in code bases. When a new software project arrives, or an existing project receives updates, the clone detection process has to be restarted. It is preferred to initially store a large amount of code from online sources permanently in a database once, and update them regularly. The clone queries can be done at any time and as many times as needed.

Fourth, online code snippets may not be a complete method or code block. We found that many answers in Stack Overflow posts contained just a fragment of code, which is not even parseable. This hinders code similarity measurement

approaches that rely on complete code structure, e.g., tree-based or graph-based approaches. Hence, some of the existing code similarity techniques do not work in this situation.

1.1 Problem Statements of the Thesis

The thesis covers several interconnected topics surrounding code similarity and code clones in large-scale source code data. The problems that this thesis tackles are discussed below.

There are several studies on issues of code cloning *from* Stack Overflow to software projects, such as security vulnerability [Acar et al., 2016], low-quality code examples [Abdalkareem et al., 2017, Zhang et al., 2018], or license violations [An et al., 2017]. However, there is limited work in the other direction of cloning, i.e., code cloning from software projects *to* Stack Overflow. The thesis asks the following questions.

Why and how code snippets appear on Stack Overflow?

What are the issues from cloned code snippets on Stack Overflow?

To offer a service for detecting code cloning to and from online sources such as Stack Overflow or GitHub, we need a tool with scalability. Based on the literature, index-based techniques offer high scalability [Hummel et al., 2010, Sajnani et al., 2016], but still suffer from low detection accuracy on clones with challenging modifications such as added, deleted, or relocated statements (type-3 clones) [Svajlenko et al., 2014b]. The thesis asks the question.

Can we improve index-based clone search techniques to find clones with challenging code modifications while at the same time preserve scalability?

On the evaluation side, the existing studies of comparing code similarity analysers contain a small set of tools, mostly clone detectors, e.g., Bellon et al. [2007], Roy and Cordy [2009a], Svajlenko and Roy [2014, 2016]. There are several tools and techniques besides clone and plagiarism detectors, such as string

matching, information retrieval, and compression methods that can locate similar code fragments. A broader comparison of code similarity analysers would be useful not only for clone and plagiarism detection study but also for studies that are based on code similarity. The thesis asks

How can we build a framework to fairly compare any code similarity tools and techniques on the same data set?

How well the code similarity tools perform compared to each other on such a framework?

Lastly, several modern code clone detection tools still do not have high recall (i.e., the number of retrieved clones over all the clones in a code corpus) on clones with a lot of modifications [Svajlenko et al., 2014b]. Kononenko et al. [2014] perform a study using code compilation as a code normalisation process to increase clone detection recall. They show that additional clones are found by the approach. We take one step further and ask, along the same lines, a question:

Can we use compilation and decompilation to improve recall in code clone detection?

To answer these questions, the thesis embarks on a journey combined with several empirical studies and evaluations of code similarity detection techniques, as will be discussed later in this thesis.

1.2 Goal and Objectives

The goal of this thesis is *to study code cloning in large-scale source code data and develop a scalable clone search approach to address challenges from such cloning.* To achieve the goal, the following objectives are set.

1. To study the problems of online code cloning between Stack Overflow and open source projects. By performing this empirical study, we can gain insights into how programmers clone code between online sources and the ramifications of doing so.

2. To create a general framework for comparing code similarity tools and clone search techniques. With the framework, we can use it to study the strengths and weaknesses of the state-of-the-art code similarity tools. The framework can also be used as a benchmark to evaluate future code similarity tools.
3. To propose an enhancement to code clone detection using compilation and decompilation.
4. To invent a scalable code clone search approach that facilitates the study of online code cloning and other studies related to large-scale code similarity detection.

1.3 Contributions

The contributions of this thesis are:

1. This thesis establishes the existence of online code clones and classifies code snippets that have been cloned to Stack Overflow into seven patterns of online code cloning.
2. The thesis studies two issues from online code clones on Stack Overflow including outdated code and license violations. The findings show an urgent need for online clone detection, which focuses on code that is cloned to and from websites instead of software.
3. This thesis presents OCD, a framework for a fair comparison of code similarity analysers on a data set with pervasive code modifications. The complete ground truth provided by the framework allows an evaluation of traditional error measures such as precision and recall, and query-based measures such as precision-at-n, average r -precision or mean average precision.
4. The thesis compares 34 code similarity analysers, the largest number of tools to date. This comparison is possibly a valuable guideline for software engineers and researchers to select the right code similarity analyser for their tasks at hand, and how to tune them to gain optimal performance.

5. This thesis shows that compilation/decompilation can be used as an effective code normalisation for clone detection. This is the first study that detects clones after compilation/decompilation. The findings show that more clones can be detected by compiling and decompiling software projects before performing code clone detection.
6. This thesis develops and evaluates Siamese, an incremental and scalable code clone search system. Siamese is suitable for code clone search on a large-scale source code data with high precision and recall.
7. The thesis demonstrates potential applications of Siamese to empirical software engineering studies.

1.4 Thesis Organisation

Chapter 2 surveys the literature on code similarity. It begins by analysing the publications in the 10-year period from 2008 – 2017 and moves to discuss in detail three research areas involving code similarity including code clones, software plagiarism, and software license compatibility. Then, it explains existing code similarity detection techniques. The chapter ends by presenting existing benchmarks for comparing code similarity analysers and discussing scalable code similarity and code search techniques.

Chapter 3 presents a study of awareness of Stack Overflow developers to the issues of outdated and license-violating code snippets via two online surveys. It explains how the online surveys are conducted and discuss the results. The findings show that several code snippets on Stack Overflow are cloned from software projects or external sources and may be problematic for reuse.

Chapter 4 confirms the findings from the Stack Overflow online surveys. It performs an empirical study of online code clone detection between Stack Overflow and open source projects. The chapter establishes evidence of code cloning to and from Stack Overflow and studies the ramifications of such cloning. It shows that an automated tool that can check for code clones from Stack Overflow or other online code corpora is essential.

Chapter 5 introduces the OCD framework for comparing code similarity analysers and explains its structure. The chapter performs an empirical study by using the framework to compare 34 state-of-the-art code similarity analysers on several scenarios of code modifications. Besides being a valuable guideline for future code similarity studies, the chapter reveals that a few string similarity techniques offer comparable results to dedicated code similarity tools, which is a useful insight we adopt for the development of our scalable clone search tool.

Chapter 6 complements the findings in Chapter 5 about using compilation and decompilation to enhance code similarity detection. The chapter extends the technique to three real-world Java projects and shows that it helps the tool to find more clones. Although the approach is useful to code clone detection in general, we do not adopt it for our scalable clone search tool due to its restrictions to only compilable code.

Chapter 7 explains the architecture of Siamese, a scalable and incremental code clone search engine incorporating multiple code representations and query reduction for an accurate clone retrieval. The chapter evaluates the accuracy of Siamese on the OCD framework compared to seven state-of-the-art clone detectors and evaluates the scalability of Siamese on a large data set of 25,000 Java projects. The chapter ends by showing that the tool's incremental update can tremendously save time when the code index needs updates.

Chapter 8 demonstrates how Siamese can be used in code similarity research. It shows three applications including online code clone detection, checking for software license compatibility of clones, and reusing of test cases. The chapter ends by showing a web-based version of Siamese.

Chapter 9 proposes the future work and concludes the thesis.

Chapter 2

Literature Survey

This chapter provides the background on code similarity, which is a fundamental concept underpinning several applications in software engineering research. The chapter presents the latest trend of code similarity research by analysing the number of publications involving code similarity in major software engineering venues in the past ten years (2008–2017) and discusses in detail three research areas that strongly involve code similarity: code clones detection, software plagiarism, and software license violations. Then, a wide range of techniques and tools to measure code similarity are explained in brief detail, including the benchmarks to compare their performance. The chapter moves to discuss emerging techniques to measure code similarity in a large-scale source code corpora. It then describes code search and clone search techniques that efficiently locate relevant code fragments based on querying a code index.

2.1 The Spectrum of Program Similarity

Similarity of computer programs¹ can be measured at three different levels. As depicted in Figure 2.1, Zhang et al. [2012] explain that two programs can be similar at the level of purpose, algorithm, or implementation. To give a simple example, two programs that sort numbers in ascending order are similar at the purpose level. They are similar at the algorithm level if they share the same sorting algorithm. Lastly, if the two programs decide to implement the algorithm in the same way, they are also

¹We use the term “program” and “software” synonymously in this thesis with the same meaning of a collection of coded instructions to perform a specific task on a computer.

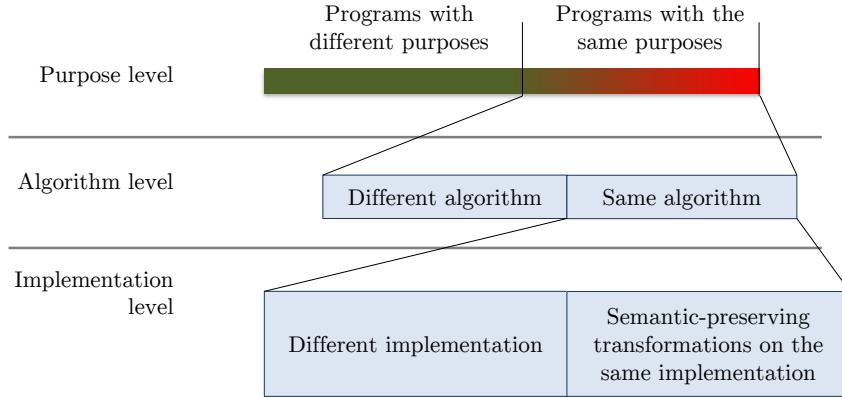


Figure 2.1: Spectrum of program similarity [Zhang et al., 2012]

similar at the level of implementation. Nevertheless, judging if two programs share the same purpose or the same behaviours (i.e., algorithm) is very difficult or even undecidable in general [Koschke, 2007]. Thus, most of the established software similarity measurements have been performed at the implementation level.

Measuring software similarity at the implementation level can also be achieved on different representations of the software: source code, compiled code, syntax tree, or software metrics extracted from the software. In this thesis, we focus on assessing similarity of software based on its source code. Hence, when we use the term “code similarity”, we mainly refer to software similarity at the implementation level represented by the software’s source code.

Code similarity is adopted in several research areas and is named differently, such as code clones, software plagiarism, copy-and-paste code, similar code, code duplication, or software redundancy. Nonetheless, the key idea is the same.

2.2 Recent Publications in Code Similarity

Code similarity has always been crucial to software engineering research with consistent numbers of publications in major venues. We studied the trend of academic publications in code similarity by collecting the papers in the past ten years (2008–2017) from eight highly-respected conferences and journals in software engineering. The venues included the International Conference on Software Engineering (ICSE), the joint meeting on Foundations of Software Engineering (ESEC/FSE), the international conference on Automated Software Engineering

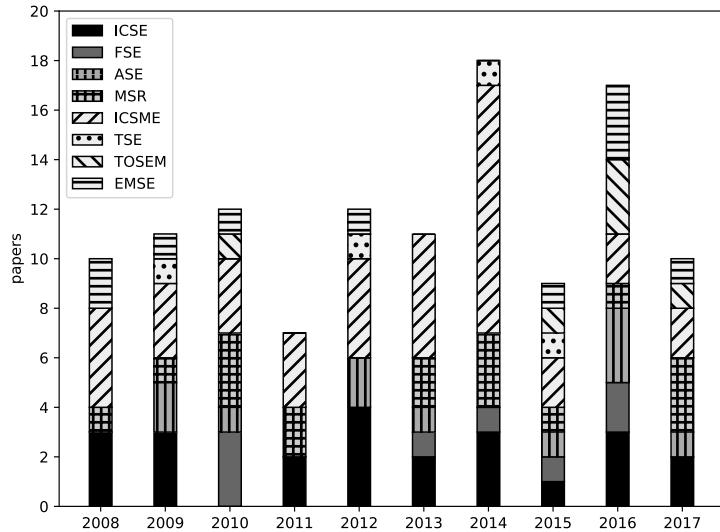


Figure 2.2: Publications relating to code similarity in major software engineering venues in the past 10 years

(ASE), the International Conference on Mining Software Repositories (MSR), the International Conference on Software Maintenance and Evolution (ICSME), the IEEE Transactions on Software Engineering (TSE) journal, the ACM Transactions on Software Engineering and Methodology (TOSEM) journal, and the Empirical Software Engineering (EMSE) journal. The paper collection was performed on **dblp computer science bibliography** (i.e., <http://dblp.uni-trier.de>) using the searches of keywords “clone/cloning”, “similar”, “duplicate/duplication”, “copy”, “redundant/redundancy” in their respective conference proceeding and journal issue pages. Then, we manually checked the papers to confirm their relevance to code similarity. We found 116 papers in total. A breakdown in years and venues is shown in Figure 2.2. As we can see from the distribution of papers in the chart, the concept of code similarity consistently appears in software engineering publications across several conferences and journal from 2008 to 2017. The majority of the publications (103 out of 116) focuses on code clones and their effects on software quality. The highest number of code similarity papers spikes in 2014, especially in ICSME which contains ten papers about code clones.

The topics discussed in the publications can be categorised into eight groups (as visualised in Figure 2.3): code similarity and clone detection techniques, clone

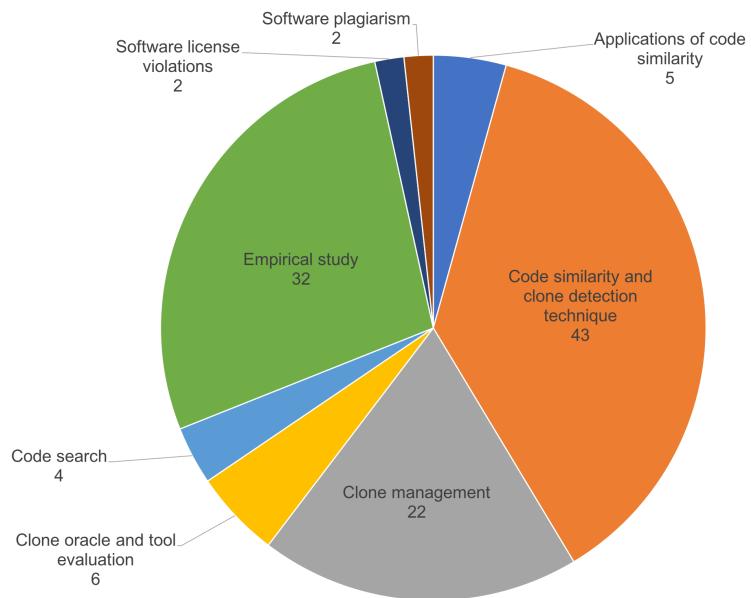


Figure 2.3: Topics of code similarity publications

management, clone oracle and tool evaluation, code search, empirical study of code clones, software license violations, software plagiarism, and other applications of code similarity. The highest number of papers fall into code similarity and clone detection techniques (43), followed by empirical studies of code clones (32), and clone management including refactoring and maintenance of clones in software (22). Although code cloning is a dominant topic in the analysed literature, we also found topics of software plagiarism (2), software license violations (2), code search, and other applications of code similarity (5) such as identifying bugs, automated transplantation, establishing software traceability, and software testing.

We select some papers from the 116 retrieved papers to discuss in this chapter. Nonetheless, they are not the only literature we reviewed, several other relevant publications from other sources are also discussed in this chapter and other chapters according to their relevance to the topic.

2.3 Research Areas Involving Code Similarity

Here, we discuss three research areas that involve code similarity in length. We start with code clones, then move to software plagiarism. Then, we present software license violations, a side-effect of code cloning. Finally, we end the discussion by

briefly explaining the other applications of code similarity to software engineering.

2.3.1 Code Clones

Code clones are fragments of code that are similar. Code clone study is an active research field in software engineering and is a dominant application of code similarity. Code cloning occurs by programmers duplicating source code with or without modifications [Roy et al., 2009] and it is a common activity found in software development. Amount of clones in software may be used as a proxy to measure software quality [Fowler, 1999]. We refer the reader to a survey by Roy and Cordy [2007], Koschke [2007], Roy et al. [2009], and Rattan et al. [2013] for comprehensive discussions on software clone detection research.

The intention behind code cloning can vary from unintentional use of coding idioms [Kapser and Godfrey, 2006] to reusing well-written code in order to preserve functionality and performance [Kamiya et al., 2002]. Software development industry has utilised code cloning intensively. Roy et al. [2009] and Davey et al. [1995] reported that a substantial percentage (7–23% and 20–30% respectively) of a software module contains clones. Baxter et al. [1998] similarly found that on average 12.7% of code in the commercial software project used in their study are clones. Interestingly, the number may increase dramatically during the past decade due to the rise of the Internet, which enables fast access and sharing of source code. A recent large-scale study by Lopes et al. [2017] shows that 70% of the code on GitHub are clones.

2.3.1.1 Code Clone Definitions

There are several definitions of when two code fragments become clones. Kamiya et al. [2002] state that code clones come from two portions of code that are either “identical” or “similar”. Baxter et al. [1998] gives a stricter definition of clones as two program fragments that are identical to each other, and call two fragments which are not identical as “near-miss clones.” Li et al. [2006] use the term “copied-pasted code” instead of clones to explain duplicated code segments. Bellon et al. [2007] refer to a clone pair as “*two code fragments form a clone pair if they are*

similar enough according to a given definition of similarity.” Roy et al. [2009] describe code clones as a result of “reusing code fragments by copying and pasting with or without minor adaptation.” An alternative definition of clone pair from Jiang et al. [2007a] is “two code fragments having similar tree representation to some level of similarity”.

Since the term “similarity” that appears in many of the clone definitions above has been independently defined by researchers, it is generally varied and open to interpretation by each particular clone detection method. The unit of measurement (i.e., code fragment, code snippet, or code portion) is also vague. Two code fragments can be either lines of code [Harris, 2003], a code block between two braces (i.e., { and }) [Cordy and Roy, 2011, Roy and Cordy, 2008, Li et al., 2006, Sajnani et al., 2016], a function [Cordy and Roy, 2011, Roy and Cordy, 2008, Jiang et al., 2007a], or the whole program [Zhang et al., 2012, Carzaniga et al., 2015]. As a result, each tool and technique treats code fragments differently based on suitability to their algorithms.

2.3.1.2 Code Clone Terminology

In this thesis, we use the following well-accepted terminology [Roy et al., 2009, Sajnani et al., 2016, Bellon et al., 2007, Roy et al., 2009, Davey et al., 1995, Carter et al., 1993] regarding code clones.

Code fragment is a segment of code represented by a triple consisting of the source file, the starting and the ending line.

Clone pair is a pair of code fragments and an associated type of similarity, i.e., Type-1, -2, -3 or -4.

Type-1 clones are literally identical code fragments except for differences in formatting such as white spaces, layouts, and comments (as shown in Figure 2.4).

Type-2 clones are syntactically identical code fragments except for differences in identifiers, literals, types, and formatting (as shown in Figure 2.5).

Type-3 clones are similar fragments with modified, relocated, added, or removed statements (as shown in Figure 2.6).

Type-4 clones are code fragments that may not be syntactically similar but

```

/* Clone#1 */
private int[] sort1 (int[] n) {
    for (int i=n.length-1; i>=0; i--)
        for (int j=1; j<=i; j++) {
            if (n[j] < n[j-1]) {
                int tmp = n[j-1];
                n[j-1] = n[j];
                n[j] = tmp;
            }
        }
    return n;
}

```

```

/* Clone#2 */
private int[] sort1 (int[] n) {
    for (int i=n.length-1; i>=0; i--)
        for (int j=1; j<=i; j++) {
            if (n[j] < n[j-1]) {
                int tmp = n[j-1];
                n[j-1] = n[j]; n[j] = tmp;
            }
        }
    return n;
}

```

Figure 2.4: Type-1 clone pair with only differences in formatting

```

/* Clone#1 */
private int[] sort1 (int[] n) {
    for (int i=n.length-1; i>=0; i--)
        for (int j=1; j<=i; j++) {
            if (n[j] < n[j-1]) {
                int tmp = n[j-1];
                n[j-1] = n[j];
                n[j] = tmp;
            }
        }
    return n;
}

```

```

/* Clone#2 */
private double[] sort2 (double[] arr) {
    for (int i=arr.length-1; i>=0; i--)
        for (int j=1; j<=i; j++) {
            if (arr[j] < arr[j-1]) {
                double temp = arr[j-1];
                arr[j-1] = arr[j]; arr[j] = temp;
            }
        }
    return arr;
}

```

Figure 2.5: Type-2 clone pair with different data types, variables and formatting

share the same semantic. Figure 2.7 shows a Type-4 clone pair of two sorting algorithms that are implemented independently. They are syntactically different but they share the same semantic based on input and output.

Clone class (i.e., clone group, clone cluster) is a set of code fragments that every two elements in the set form a clone pair.

```

/* Clone#1 */
private int[] sort1 (int[] n) {
    for (int i=n.length-1; i>=0; i--)
        for (int j=1; j<=i; j++) {
            if (n[j] < n[j-1]) {
                int tmp = n[j-1];
                n[j-1] = n[j];
                n[j] = tmp;
            }
        }
    return n;
}

```

```

/* Clone#2 */
private int[] sort3 (int[] arr) {
    int i=arr.length; int j=1;
    while (i < arr.length) {
        while (j < i) {
            if (arr[j] < arr[j - 1]) {
                int temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
            j++;
        }
        i--;
    }
    return arr;
}

```

Figure 2.6: Type-3 clone pair with modified/added/deleted/relocated statements

```

/* Clone#1 -- Bubble sort */
private int[] sort1 (int[] n) {
    for (int i=n.length;i>=0;i--) {
        for (int j=1;j<=i;j++) {
            if (n[j]<n[j-1]) {
                int tmp = n[j-1];
                n[j-1] = n[j];
                n[j] = tmp;
            }
        }
    }
    return n;
}

/* Clone#2 -- Insertion sort */
private static int[] sort4(int[] n) {
    ArrayList<Integer> s =
        new ArrayList<Integer>();
    for (int i = 0; i < n.length; i++) {
        for (int j = 0; j < s.size(); j++) {
            if (n[i] > s.get(j)) {
                s.add(j + 1, n[i]);
                break;
            }
        }
    }
    return n;
}

```

Figure 2.7: Type-4 clone pair with clearly distinct code structure but share the same semantic based on input/output. The clone fragment on the left implements bubble sort algorithm while the clone fragment on the right implements insertion sort.

Syntactic clones: We may call Type-1, Type-2, and Type-3 clones as syntactic clones because they mostly preserve the semantic of the originals while differ at syntactic level [Kim et al., 2011]. Syntactic clones are commonly found in software systems as suggested by empirical clone studies. Li et al. [2006] discovered that identifier renaming (Type-1 and Type-2) accounts for 65–67% of clones residing in Linux and FreeBSD, and 23–27% of cloned fragments contain inserted, deleted, or modified statements (Type-3). Svajlenko et al. [2014a] analysed IJaDataset 2.0 [ASE group, 2018], a large Java data set (25,000 subject systems, 365M SLOC), and found 6.2 million clone pairs of Type-1 to Type-3. Interestingly, 6.1 million pairs are Type-3.

Semantic clones: For Type-4 clone pairs, they may not resemble the same syntax and only contain an equivalent program semantic. Hence, they are sometimes called semantic clones [Funaro et al., 2010]. Since program equivalence is generally undecidable, detecting Type-4 clone has to be bounded within subsets of program behaviours, such as measuring semantic similarity of two pieces of code based on pre- and post-conditions [Bellon et al., 2007], or measuring similarity of core values in their executions [Zhang et al., 2012] or over multiple states from an execution [Carzaniga et al., 2015]. Semantic clones also cover the problem of plagiarism at an algorithmic level [Zhang et al., 2012].

Simions are Type-4 clones with a stricter definition. They must be created

independently without copying and pasting from another code [Juergens et al., 2011].

2.3.1.3 Are Clones Good or Bad?

The question of “*are code clones good or bad?*” have been a debated problem in clone research community for quite a long time. Clones are initially believed to be one of code smells, which increase complications in project maintenance [Fowler, 1999]. This is because code cloning increases a software project size and, at the same time, increase the cost to maintain the software. If a fault is found in a single instance of cloned code, all cloned fragments have to be located and updated [Kamiya et al., 2002, Bellon et al., 2007]. Li et al. [2006] has found 49 bugs in Linux and 31 in FreeBSD originating from clones. Moreover, duplicated code exists because of lack of procedural abstraction or inheritance [Ducasse et al., 1999]. Juergens et al. [2009] empirically showed that inconsistent changes in code clones could lead to faults by analysing four industrial and one open source projects. A large-scale study by Mondal et al. [2018] using two clone detectors on twelve software systems in Java, C, and C#. They analysed the clones using eight code stability metrics adopted from the previous studies and found that cloned code is less stable than non-cloned code.

On the contrary, several empirical clone studies have shown that clones are not always bad. Aversano et al. [2007] and Thummalapenta et al. [2010] report that most of their detected code clones are changed consistently, or they evolve independently. Kapser and Godfrey [2006, 2008] show that code clones are not always harmful and can be beneficial in several cases, such as when building software to support a set of hardware or platforms. Krinke [2008] argues that cloned code is more stable than non-cloned code, based on number of changes to the systems, in his empirical study of clone evolution of five software systems over 200 weeks. Along the same lines, Rahman et al. [2010, 2012] argue that the belief of code smells originating from code clones are not entirely accurate. Their empirical study on four C open source projects over ten thousand snapshots show that most of code clones do not cause bugs in software. 80% of bugs in three

projects contain no cloned code at all and larger clone groups do not associate with more bugs than smaller ones. Code with clones may even contain fewer defects than the one without clones. Göde and Harder [2011] studied change frequency of clone genealogies over revisions of three software projects and found that 87.8% of the clones were rarely changed (never or once). Among the changed clones, only 3% contained high-severity problems. A large-scale study of 4,421 open source Java projects by Saini et al. [2016b] found no statistically significant difference in code quality between cloned and non-cloned code for 24 out of the selected 27 software metrics.

It seems, at least for now, that there is no definite answer to the question “*are code clones good or bad?*” Code clones may or may not introduce complications into software maintenance based on several contexts, such as the languages and types of software projects being analysed (e.g., one version vs. multiple versions of hardware drivers) [Kapser and Godfrey, 2006] or how consistent the changes are applied to the clones [Aversano et al., 2007, Juergens et al., 2011]. Nonetheless, clone researchers agree that **we need to make clones in software explicit** so that an appropriate clone management process can be carried out [Chatterji et al., 2016].

2.3.1.4 Clone Management

The second largest portion (19%) of code similarity research in the past ten years is clone management, i.e., how to maintain and reduce risks from clones within software projects. A survey of clone research community [Chatterji et al., 2016] shows that researchers believe that clone management is useful for maintenance tasks which affect long-term system quality. Kim et al. [2005] studied clone genealogies in two Java projects and reported that many clones are not long-lived and aggressive clone refactoring may not always be beneficial. Duala-Ekoko and Robillard [2008] create CloneTracker, an Eclipse plug-in, to support programmers in monitoring clones in a software project. Wang et al. [2012] take a preventive approach by predicting the harmfulness of clones at a time of cloning using machine learning techniques. Nguyen et al. [2012] create a comprehensive clone management tool called JSync that offers code clone detection, code consistency

validation, clone synchronising and clone merging. Zhang and Kim [2017] reuse tests among clones to detect bugs from inconsistent clone updates. Their approach automatically transplants non-tested clone fragments in place of another clone fragment with a test case to test them.

Some researchers focus on clone refactoring. Bazrafshan and Koschke [2013] discovered that clone removals happened both deliberately and accidentally. They also found that some clone refactorings were not complete and a clone detection is needed for checking of non-refactored clones. Tsantalis et al. [2015] assess a possibility of clone refactoring by using nesting structure tree (NST) matching and statement mapping based on program dependence graph. Similarly, Wang and Godfrey [2014] develop an approach to recommend clones for refactoring based on a decision tree classifier. There are a few automated techniques introduced to facilitate clone refactoring. Li and Thompson [2008, 2011], and Brown and Thompson [2010] present semi-automatic clone refactoring tools with programmers in-the-loop for Erlang and Haskell. Tsantalis et al. [2017] use Lambda expressions to refactor Java clones automatically. Lin et al. [2014] present an approached called MCIDiff (Multi-Clone-Instances Differencing) to show differences among instances of clones in the same clone group to the programmers. MCIDiff Eclipse plug-in shows high precision and recall of clone difference summary, which helps facilitating clone refactoring decision.

2.3.1.5 Patterns of Code Cloning

Kapser and Godfrey [2003] performed an empirical study of code clones in subsystems of Linux file system to understand why code clones happen, and how they are created. They defined seven types of clone taxonomy: 1) copied code blocks in the same function, 2) comparable functions in the same file, 3) copied functions between files within the same directory, 4) copied functions between files across different directories, 5) whole-file clone (possibly with some changes), 6) code blocks across files, and 7) need for initialisation and finalisation code fragments of data or function.

Later, they performed a follow-up study [Kapser and Godfrey, 2006, 2008] by

Group	Pattern	Description
Forking	Hardware variation	Cloning to create a new driver for a hardware family
	Platform variation	Cloning to port software to a new platform
	Experimental variation	Cloning to experiment with pre-existing code
Templating	Boiler-plating	Cloning due to language inexpressiveness
	API/library protocols	Cloning of an ordered set of API-related procedure calls
	General language/ algorithmic idioms	Cloning of well-written solutions
Customisation	Parameterized code	Cloning by changing identifier names
	Bug workarounds	Cloning and fixing a bug (when one cannot modify the original code)
	Replicate and specialise	Cloning and adapting of code
Exact matches	Cross-cutting concerns	Cloning of some functionalities across different locations (e.g., logging and debugging)
	Verbatim snippets	Cloning of small repetitive fragments of logic (e.g., branching control)

Table 2.1: Code cloning patterns by Kapser and Godfrey [2008]

analysing three systems for recurrence of cloning behaviours and discovered eleven patterns of code cloning, which are categorised into four main groups, as listed in Table 2.1. The patterns explain why or how code clones are created. The four main groups include forking, templating, customisation, and exact matches.

2.3.2 Software Plagiarism

Another well-established application of code similarity is detecting software plagiarism. While the term “plagiarism” is commonly used in the context of written text in natural languages such as reports, essays, theses, or conference and journal papers, it can be applied to any kind of data [Clough, 2000]. In computer science, plagiarism occurs in program source code [Cosma and Joy, 2008]. Unfortunately, automated plagiarism detection tools for natural text such as Turnitin [2015] cannot perform well on detecting plagiarised source code [Weber-Wulff et al., 2012]. Due to inherent differences in source code and natural text, one cannot just reuse the detection method for natural text on source code.

Plagiarism of programming assignments has been occurring in higher education, such as in a university, for several decades. There were attempts to alleviate

this problem since the 1970s [Ottenstein, 1976, Donaldson et al., 1981, Grier, 1981, Berghel and Sallach, 1984]. A survey by Cosma and Joy [2008] found that approximately 50% of UK academics who taught at least one programming course believe that an act of copying source code from someone else and submitting as their own (with or without changes) without an acknowledgement is plagiarism. Another survey [Daniela et al., 2012] from students and faculty staffs shows that students' work have been plagiarised at least once and the plagiarised documents are either computer programs or homework assignments. Most of the students (72%) state that they use solutions from their peers to inspire their answers and only 10 percent of the surveyed students never look at solutions from their classmates. However, more than 20 percent of the staffs do not want to perform any plagiarism checking, and 12 percent of the staff carry out no checking at all. More than half of the staffs (72%) admit that an automated tool can raise their incentive of plagiarism check in programming submissions. The best solution seems to be a combination of an effective grading method plus an efficient detection tool.

Similar to education, plagiarism in commercial computer programs has been occurring for decades. Dated back in the 1980s, there were a few lawsuits regarding plagiarism of computer programs. One example is a case between SAS Institute and S&H Computer Systems [SAS Institute, Inc. v. S&H Computer Systems, 1985]. SAS Institute sued S&H Computer Systems for copying its statistical analysis program *SAS* running on IBM computers. A similar case occurred between Whelan Associates and Jaslow Dental Labor [Whelan Associates v. Jaslow Dental Labor, 1985]. Since then, the problem of plagiarism in commercial software persists until the present, e.g., Computer Associates Int'l, Inc. v. Altai, Inc. [1992], Compuware Corp. v. International Business Machines (IBM) [2002], Oracle America, Inc. v. Google Inc. [2012].

Due to an early age of computer law in the 1990s, the problem of software copyright was controversial. In the United States, Hamilton and Sabety [1997] describe that the court defines copyright of software to cover both literal copy and non-literal copy. Literal copying of user interface, source code, and object code

infringes copyrights, so any attempt to copy and reuse them causes issues. On the contrast, the scope of non-literal copying is controversial. They suggest that algorithms, data structures, hardware drivers, and programming languages are not copyrightable. Nonetheless, a data processing component including a collection of data structures and algorithms to achieve a specific task by the software is copyrightable but has to be scrutinised with strong computer science concepts in mind.

Consequences of software plagiarism have caused IT companies to suffer from revenue loss due to their products being copied. In a global landscape, the cost of global PC software piracy in 2010 is as high as 58.8 billion dollars [Business Software Alliance (BSA), 2011]. One of the prominent cases is Oracle America, Inc. v. Google Inc. [2011]. Oracle sued Google in August 2010 for violating their copyright of Java APIs by copying parts of code from 12 source files and 37 Java specifications and use them to develop the Android operating system. Google defended that the copied code was too minimal, or “de minimis”, to be counted as copyright infringement [Sprigman, 2015]. However, as of now, this case is still in progress, and Oracle declared enormous damage of nine billion dollars [Jeong, 2018].

Plagiarism can also occur in mobile applications. With the rapid growth of mobile phone usage, the number of mobile applications has been increasing sharply. There are approximately more than one million applications available in Apple App Store and Google Play store in 2014 [Adjust, 2014, Viennot et al., 2014]. This vast number attract plagiarisers to reverse engineer mobile applications by decompiling an original app (especially Android), make alterations, repackage, and submit it to the same or different app store for malicious purposes. The worst of all, normal applications are converted to malware [Crussell et al., 2013, Zhang et al., 2014, Zhou and Jiang, 2012]. Chen et al. [2014] discovered that 13.51% of applications from five different Android markets in China, Europe, and America are clones. These clones can divert 10% of user and 14% advertisement viewing rate from the original which may generate a substantial drop in revenue to the real application

owners [Gibler et al., 2013].

2.3.3 Software License Compatibility

Software licenses strongly involve code similarity. There are studies focusing on software licensing conflicts among clones [An et al., 2017, German et al., 2009]. Since each software project has its own license which may be different from another, a careless code cloning from one project to another possibly creates a license conflict, which produces serious legal consequences [Gross, 2007, Lee, 2008, Hemel et al., 2011, Economy, 2009].

A study of software licensing evolution among six free and open source software systems (FOSS) [Di Penta et al., 2010] shows that a change of software license, i.e., license evolution, affects software usage in both positive and negative way. It can prohibit users from reusing the software due to licensing incompatibility. On the other hand, it also sometimes encourages distribution and reuse of a software system and increasing numbers of developers involved. An empirical study of code siblings (i.e., clones among different systems that come from the same source) [German et al., 2009] reported up to 2,208 siblings between BSD kernels and Linux kernel with different licenses.

There are several automated tools for software license identification available, such as FOSSology, Ohcount, Open Source License Checker (OSLC), and Ninka. *FOSSology* [Gobeille, 2008] deploys Binary Symbolic Alignment Matrix (bSAM) pattern matching algorithm to match software licenses. *Ohcount* [Verprauskus, 2016] is a code line counter that also detects software license with regular expressions. *OSLC* [Oksanen and Kupsu, 2016] tool stores a database of software licenses and matches license from source code files in a subject software system by using Heckel's algorithm of isolating textual differences [Heckel, 1978]. *Ninka* [German et al., 2010] uses a sentence-matching method for license identification, which outperforms the other three tools. The authors applied Ninka to analyse license of 0.8 million source code files in Debian and found that GPLv2+ was the most popular license.

2.3.4 Other Applications of Code Similarity

The applications of code similarity are not only limited to code clone and plagiarism detection but also span to several research areas under the umbrella of software engineering. Here, we discuss a few selected applications based on code similarity.

2.3.4.1 Automated Program Repair

Code similarity can be exploited to find candidates for program repair. Carzaniga et al. [2015] present a technique for a self-healing system that automatically fixes itself when a failure occurs by replacing a buggy part of code by another redundant and non-failing alternative. They propose a model to measure software redundancy by computing similarity of action logs derived from executions of two code fragments. Code fragments that are redundant in their behaviours are useful in building a self-healing system. Barr et al. [2014] present and validate the concept of “plastic surgery hypothesis”, meaning that one can write new code just by reusing and combining existing code statements from the same codebase. This hypothesis supports the ideas of automated program repair and genetic improvement that reuses existing code to fix a problematic portion of a program or to adapt existing code to match with new specifications. The experiment shows that many commits can be recreated from existing code. Interestingly, 30% of code changes from commit logs can be assembled from existing code in the same file, and 9% are from within the same package. Ke et al. [2015] search for semantically similar code from a code database using a search query of input-output specifications extracted from an existing code fragment. The authors use this code search technique to find repair candidates of faulty programs. They show that a tool incorporating semantic code search combined with automatic program repair techniques, called *SearchRepair*, fixes 97.3% of bugs in six programs in their study.

2.3.4.2 Finding Redundant Implementations

Kawrykow and Robillard [2009] use code similarity to find cases of API imitation (i.e., writing the same code that is already provided by the APIs) and perform a study on ten Java projects. They rely on an abstraction of a method body (i.e.,

a set of the fields, methods, and types that are called by the method) to find a match between methods in a software project and methods in APIs. They found 400 cases of API imitations in 10 popular Java open source projects. By removing the imitations, the developers could improve the quality of the software by saving a large number of methods calls. Bauer et al. [2014] introduce a technique to find unintentional re-implementations based on similarity of identifiers in Java projects. The proof-of-concept tool found seven cases of re-implementations in three Java open source projects. The authors later extended the concept to use latent semantic indexing (LSI) and code clone detector (ConQAT) [Bauer et al., 2016]. They found that the two techniques complemented each other and the detected re-implementations were considered relevant by practitioners.

2.3.4.3 Software Test Improvement

Carzaniga et al. [2014] utilise redundancy among Java methods to improve software testing. Redundant methods are used to create cross-checking oracles, i.e., check the validity of tested code by replacing a method call by a call to its semantically similar method. Jalbert and Bradbury [2010] propose a way to find important bug patterns in concurrent software by performing code clone detection between software and manually-created code fragments of bug patterns.

2.4 Existing Code Similarity Detection Techniques and Tools

A significant amount of work in code similarity is dedicated to techniques for locating similar pieces of code and their applications to software developments. Since the 1970s, there are myriad of tools and techniques created for code similarity measurement. The comprehensive lists of tools and techniques can be found in the papers by Roy and Cordy [2007], Koschke [2007], Rattan et al. [2013] and Ragkhitwetsagul et al. [2018a]. Code similarity detection tools employ different approaches to detect similar code fragments inside a program or among programs. Similar to a survey of clone detection techniques by Roy et al. [2009], the approaches and tools for code similarity presented in this thesis are categorised

into different groups based on the level of abstraction of the source program, such as metric-based, text-based, token-based, tree-based, or graph-based techniques. Since Roy et al.'s survey is from 2009, we also add more tools and techniques that are later introduced after the survey such as compile code-based and model-based techniques. Moreover, we augment the list by including techniques from other research areas (e.g., Kolmogorov complexity, Software Bertillonage, and information retrieval). Some tools are hybrid, i.e., they are a combination of several approaches, so they will be discussed in multiple sections related to each specific part of their techniques in this chapter.

2.4.1 Metric-based Approaches

Metric-based approaches, such as software measures, are used to compute software similarity in the early years of development in code similarity measurement. Unfortunately, because of its superficial measurement and lack of structural understanding of programs, later it has been overshadowed by newer and more sophisticated techniques. We discuss a few metric-based approaches in this thesis as follows.

One of the earliest software plagiarism detection systems by Ottenstein [1976] bases on Halstead complexity measures [Halstead, 1977]. It discovered one plagiarised pair out of 47 student submissions. Donaldson et al. [1981] continued the work using software parameters to search for copying in FORTRAN assignments by introducing eight finer-grained software metrics. Moreover, to increase accuracy, the structure of a program was captured using a sequence of statement ordering. Grier [1981] created a tool called *Accuse* that computed 20 software parameters, of which seven are critical for calculating a correlation score, to detect plagiarism in Pascal assignments. Berghel and Sallach [1984] showed that using only four Halstead's software parameters was not efficient for measuring software similarity since it produced many false positives. They introduced *program profile*, i.e., a tuple of 15 selected parameters, as a solution. Faidhi and Robinson [1987] evaluated previous approaches based on software parameters and found that the accuracy were limited. They proposed a new approach of using empirical analysis, i.e., an analysis that aimed to understand intrinsic characteristics of a subject being tested. The two

authors proposed *empirical metrics* contained 28 measures to capture both general and inherent features of a program.

Nevertheless, these metrics-based approaches are found to be less effective compared to newer approaches and are no longer suitable for locating similar code in software projects. An empirical study [Kapser and Godfrey, 2003] found that metric-based approach performed better in finding small function clones, while a more sophisticated CCFinder clone detector performed better on larger ones. Moreover, the metric-based approach could only locate very similar clones, but CCFinder was more flexible on finding clones with modifications.

2.4.2 Text-based Approaches

Text-based approaches computes code similarity by comparing two sequences of strings. It can locate clones created by copying and pasting without alteration (Type-1), while suffers from finding clones having syntactic or semantic modifications (Type-2, -3, or -4). Additional techniques have to be included in the similarity computation process to allow flexibility in detecting syntactic and semantic similarity. The methods and tools based on textual comparisons are listed below.

2.4.2.1 Longest Common Subsequence (LCS)

Longest common subsequence (LCS) [Bergrøth et al., 2000] is a string matching algorithm based on edit distance. It finds a similar segment between two strings based on the number of insertions and deletions to be made. LCS is initially proposed to compare amino acid sequences [Needleman and Wunsch, 1970], but the algorithm can generally be applied to any string comparison including source code. There are several variations of LCS algorithms, e.g., Needleman and Wunsch [1970], Hirschberg [1977], Hunt and Szymanski [1977], Apostolico and Guerra [1987].

There are a few code similarity detection tools implementing LCS. *NiCad* [Cordy and Roy, 2011, Roy and Cordy, 2008] is a clone detector that compare lines of source code. The tool employs TXL grammar [Cordy, 2006] to parse and precisely transform specific parts of code. Many phases occur during NiCad's clone

detection process including code pretty-printing, code abstraction, code normalisation, and code filtering. In the similarity computation phase, NiCad applies LCS to compare pre-processed lines of code from two programs. Furthermore, LCS is also adopted by *Plague* [Whale, 1990], *YAP* [Wise, 1992], and *CoP* [Luo et al., 2014].

Besides LCS, other string matching techniques are also adopted for computing code similarity. *Duploc* [Ducasse et al., 1999] locates duplicated code inside a software project using an undefined line-based string matching technique and visually reports the matches with a comparison matrix. *Simian* [Harris, 2003] is a clone detection tool based on line-by-line textual comparisons with abilities to detect clones having identifier renaming. Lastly, PMD’s Copy/Paste Detector [CPD] relies on string matching using the Karp-Rabin algorithm.

Most of the text-based tools discover cloned or plagiarised code fragments that are identical or very similar with only minor alterations. However, they cannot effectively detect similar code with added, deleted, or relocated statements [Cosma and Joy, 2012]. According to a study by Roy et al. [2009], most of the text-based tools can fully or partially locate Type-1 clones. However, they, except NiCad, fail to discover clones of Type-2, Type-3, and Type-4.

2.4.3 Token-based Approaches

Token-based approaches take one step of abstraction up from the source code text and convert a program into tokens. A stream or a set of the tokens are used as an abstract representation of the program. The tokens may be normalised (i.e., being replaced with a more abstract representative token) to get rid of all textual differences and to capture only the structure of the program. Several similarity measurements are then applied on the tokens. The token-based approach is the most popular approach in code similarity due to its simplicity, flexibility, and scalability in code matching. Some selected tools and techniques that rely on a token representation of source code are discussed below.

2.4.3.1 Normalised Tokens

Normalised tokens offer an advantage of overlooking changes in formatting and identifiers. The token normalisation enables cross-language clone detection because it normalises code written in different programming languages to a set of common abstracted tokens. We adopt an example of normalised tokens from Gitchell and Tran [1999] here for an illustration purpose.

A code fragment

```
for (i = 0; i < max; i++)
```

can be converted to a normalised token sequence

```
TKN-FOR TKN-LPAREN TKN-ID-I TKN-EQUALS TKN-ZERO ... TKN-RPAREN
```

where the token TKN-FOR represents the keyword `for`, the token TKN-LPAREN represents the left parenthesis `(`, the token TKN-ID represents the variable `i`, and so on. Program similarity is then computed based on this token representation. There are numerous tools based on this idea but with different definitions of tokens, and similarity measures.

Plague, a tool introduced by Whale [1990], captures program structure and finds plagiarism in programming assignments. The Plague tool creates a structure profile of programs and filters out dissimilar programs using the profiles. Then, among the remaining programs, Plague generates tokens from them. The tool compares the tokens using Heckel's algorithm [Heckel, 1978] due to its robustness to statement relocations over LCS. Joy and Luck [1999] created *Sherlock* tool and integrated it into their programming course management system [Joy et al., 2005]. Sherlock performs incremental comparison by making five different comparisons ranging from a textual to token-based comparison. *Sim* [Gitchell and Tran, 1999] utilises normalised tokens and transforms a statement into a token stream of pre-defined tokens. Then, the token streams from two programs are compared using a string alignment algorithm.

Similarly, *YAP* (*Yet Another Plague*) is a plagiarism detection tool based on tokens. The latest version, YAP3 [Wise, 1996], changes its internal algorithm

from LCS in the original YAP [Wise, 1992], and Heckel’s algorithm [Heckel, 1978] in YAP2 to *Running-Karp-Rabin Greedy-String-Tiling*. (*RKS-GST*) [Wise, 1993]. A program is parsed into tokens first. Then, all tokens are included in a tile and compared by finding a maximal match between them. *JPlag* [Prechelt et al., 2002] implements a *Greedy-String-Tiling (GST)* string comparison method similar to YAP3. It enhances precision by converting a program into a token string with better semantic meaning, e.g., BEGIN_METHOD is used in JPlag instead of just OPEN_BRACE in YAP3. Using the GST, two token strings are searched for the maximum contiguous match. The Karp-Rabin pattern matching is executed first to find substrings with the same hash value, and the GST is later applied to the results for finer-grained textual comparison.

Code Clone Finder (i.e., *CCFinder*, *CCFinderX*, *ccfx*), created by Kamiya et al. [2002], extracts tokens from an original program and transforms them to normalised tokens using predefined language-specific transformation rules, e.g., variables are replaced with special token \$. Then, a suffix tree is created from the normalised token stream, and a tree-matching algorithm is used to find clones. Besides clones, the authors introduce several supporting metrics such as a length of code portion, a population of clone class, a deflation rate from removing clones, and a radius, i.e., maximum length from each file to its top ancestor. *NiCad* [Cordy and Roy, 2011, Roy and Cordy, 2008], as previously discussed, not only utilises LCS text-based approach, but also integrates tokens in its code abstraction and code normalisation phases. The tool applies TXL grammar to convert a specific part of code into abstracted or normalised tokens before a comparison. Similarly, *iClones* [Göde and Koschke, 2009] is a token-based tool that offers a capability to locate clones over several revisions of a software system. *CP-Miner* [Li et al., 2006], a data-mining-based clone detection tool, uses tokens to avoid identifier and data type renaming before performing a clone detection using data mining algorithms. Li and Thompson [2008] use tokens and suffix trees to locate clone candidates before applying a finer-grained clone filter using an AST-based technique.

2.4.3.2 N-Gram

n-gram (*k*-gram, *q*-gram, or *k*-shingle) is a contiguous sequence of *n*-size substrings of a given string. Given a string S and a number n , $S[i, i+n-1]$ is an *n*-gram of S starting at the i -th character. The size of n can be varied and carefully chosen to suit the task. *n*-grams are suitable for partial matching in text [Li et al., 2007] and code [Schleimer et al., 2003]. It is extensively utilised in computational linguistics since it provides an ability to predict the next item in the sequence by probabilistic model [Hindle et al., 2012]. It is also adopted in code plagiarism and clone detection in combination with normalised tokens [Schleimer et al., 2003]. After a stream of tokens is created, the token stream is parsed into a stream of consecutive *n*-grams. Then, the tool performs its comparison based on the *n*-grams. Tools that utilise *n*-grams are as follows.

MOSS [Aiken, 2015], a well-known software plagiarism detector, relies on the concept of *n*-gram, and a document fingerprinting algorithm called *winnowing* [Schleimer et al., 2003]. The algorithm converts a source code string into *n*-grams, computes a hash sequence of all *n*-grams, and creates a sliding window over the sequence to choose a fingerprint. The set of fingerprints is compared with the fingerprints from other programs for similarity. Burrows et al. [2007] present a solution to software plagiarism that scales to large code repositories using a combination of *n*-grams, multiple local alignment, and an inverted index. A normalised token stream, which is derived from the original source code, is converted into a set of *n*-grams (with $n = 4$) and added into an inverted index. After the inverted index has been created, one can give a suspected source code as a query to search for similar code fragment candidates. Then, the multiple local alignment [Morgenstern et al., 1998] is executed for fine-grained similarity computation. Likewise, Smith and Horwitz [2009] present a hybrid approach of finding code clones by applying a fingerprinting technique to code blocks. *n*-grams are first created from the source code files. Then, a fingerprint, which captures unique characteristics of the program, is obtained by concatenating *k-least frequent n-grams*. The approach can report clone clusters by grouping clones that have

similarity scores within a specific threshold. It can also query clones of a given source code fragment by using rank ordering. Duric and Gasevic [2013] discover flaws in JPlag which is caused by overly abstracted and unnecessary tokens. A hybrid tool called *Source Code Similarity Detector System (SCSDS)* is proposed to fix the problems. The authors invent a new tokenising technique with finer-grained token definitions to avoid false positive results. The tool offers higher detection accuracy by combining two similarity measures: the RKR-GST and winnowing. It computes a total similarity using weighted scores from the two algorithms.

2.4.4 Tree-based Approaches

Tree-based approaches can partially handle structural code modifications when locating similar programs. Usually, an Abstract Syntax Tree (AST) is used to represent program structure. To compare with another program, tree or subtree similarity of two ASTs is computed. Tools utilising ASTs can accurately locate a specific segment of code in the program by traversing the tree. Hence, it can find location-specific cloned/plagiarised code. However, it has a significant drawback of high computational complexity. A comparison of two ASTs with N nodes can have an upper bound of $O(N^3)$ [Baxter et al., 1998]. Tree-based tools typically integrate some heuristics or optimising mechanisms to lower their computational complexity.

CloneDR [Baxter et al., 1998] is among the first that deploys tree-based techniques in clone detection by using ASTs and hashing to locate clones. The tool creates an AST from a source code file and hashes its subtrees into different buckets. The hashing reduces the number of comparisons extensively from the number of programs (N) to the number of bucket B , where $B \ll N$ (e.g., the authors choose $B = \frac{N}{10}$). Clones are discovered within each bucket using subtree comparisons. It can find near-miss clones (i.e., clones with slight modifications) by utilising a special hash function that ignores identifiers. *Deckard* [Jiang et al., 2007a] incorporates several optimisation techniques into their tree-based algorithm. To circumvent a computational obstacle of tree similarity measure, it uses a *characteristic vector* to approximate an AST, which offers much lower complexity in similarity comparison. Characteristic vectors of all candidate programs are

clustered using Locality Sensitivity Hashing (LSH) [Slaney and Casey, 2008] based on Euclidean distance. The clusters collect clones that fall within a specified similarity threshold.

Li and Thompson [2008] employ a tree-based structure called *annotated abstract syntax tree (AAST)* to filter clone candidates detected by a token-based technique. AAST encodes information of Erlang/OTP code fragments' locations and binding structure information. AASTs of the clone candidates are used to decompose the clones into small syntactic units, which are then matched for clones using consistent variable renaming. Brown and Thompson [2010] create a clone detection tool for Haskell and embed it in a refactoring framework called HaRe. The tool uses abstract syntax trees for finding clones in Haskell code bases. The tool has high precision but is not scalable due to complexity in their clone detection algorithm.

Falke et al. [2008] leverage the linear-time clone detection using suffix trees by transforming a syntax tree structure (AST or parse tree) to a suffix tree using preorder traversal. Then, post-processing is done to decompose the reported clones, which can be a segment of syntactically incomplete code, into smaller complete syntactic clones. Their suffix tree technique locates only Type-1 and Type-2 clones. An empirical evaluation shows that the technique is 60–80% faster than typical AST-based techniques.

Tree-based approaches give more flexibility of measuring syntactic and semantic code similarity over text and token-based approaches. They are robust against identifier renaming and formatting changes while capture structure of programs. They can detect Type-1, Type-2, and some Type-3 clones. A significant drawback of the approaches is the complexity of tree comparisons, making it not scalable. Thus, various optimisation techniques have been employed to improve the speed of similarity computation. Although offering some flexible matchings, tree-based approaches are still susceptible to heavy structural changes such as changing of statements, e.g., `for` to `while`, `if-else` to `case`, or heavy code block relocations.

2.4.5 Graph-based Approaches

A graph-based structure is adopted to capture the semantics of a program and ignore the programming language literals and syntaxes. Graph-based code similarity detection can cope with all minor changes in formatting, identifiers, basic block relocations, and loop- or conditional-statement replacements. Unfortunately, like trees, graphs inherently suffer from the time complexity in measuring their similarity. The algorithms for graph-based software comparison are mostly NP-complete [Liu et al., 2006, Crussell et al., 2012, Krinke, 2001, Chae et al., 2013]. Thus, optimisation techniques to circumvent this computational complexity are incorporated into the graph matching process. In clone and plagiarism detection, two specific types of graph, i.e., program dependence graph (PDG), and control flow graph (CFG) are often used to represent programs.

2.4.5.1 Program Dependence Graph (PDG)

PDG is a directed graph which captures data and control dependencies in a program. Krinke [2001] introduces an approach on finding similar code using a special type of program dependence graph (PDG) called *fine-grained program dependence graph* combining AST and PDG characteristics. A similarity measure, called *maximal similar subgraphs*, chooses a pair of starting vertices between 2 graphs and then keeps including new similar vertices and edges until reaching the limit of inclusion (k -limit). The approach relies on weighted subgraphs, which gives a higher priority to subgraphs with more data dependencies. An evaluation shows that the approach achieves high precision and recall at the same time. However, it cannot handle large projects due to its high computational time [Bellon et al., 2007]. Komondoor and Horwitz [2001] also detect clones using PDG, but with a different technique. They rely on program slicing [Weiser, 1984] to locate code clones. A program is initially converted to its PDG representation. Backward slicing is then deployed to find subgraph isomorphism based on the slices, resulting in clone fragments of two programs, and forward slicing is added to increase accuracy. The experiment shows that their approach can detect different types of clones including non-contiguous (having gaps), reordered, and intertwined clones. However, it has a serious

drawback of running time. Liu et al. [2006] create a tool called *GPLAG* for software plagiarism detection based on PDGs. A execution time limit is used to filter out unusually long graph isomorphic computations. Moreover, an optimisation by applying lossless and lossy filters helps to reduce running time. Gabel et al. [2008] tackle the computational limitation in PDG-based clone detection by converting PDG subgraphs of the program being analysed into abstract syntax trees. Then, approximate tree-based similarity using characteristic vectors, used by Deckard, is applied to detect clones. An evaluation shows that their PDG-AST technique locates more clones than using the Deckard's pure AST technique. Moreover, it scales to a large Linux code with 7M SLOC.

2.4.5.2 Control Flow Graph (CFG)

Chae et al. [2013] presents a graph-based approach to detect software plagiarism by using static analysis to extract features from a program and represent them as a graph, without a need to analyse the source code. The authors develop an API-based control flow graph (A-CFG), which shows the relationships of API calls and the sequences of calls within a program, and use *Random Walk with Restart (RWR)* algorithm to generate a score vector of each A-CFG. Finally, cosine similarity is used to calculate a similarity score between two vectors. Chen et al. [2014] use CFG to represent behaviours of a program. They find cloned Android applications based on a special type of CFG called 3D-CFG (discussed in the next section).

Graph-based approaches offer the highest flexibility to clone and plagiarism detection and produces high precision and recall. Similar to the tree-based approaches, it can narrow down the scope of detection to a specific segment of code. The graph-based approaches outperform token-based methods when it comes to highly modified code [Li et al., 2006]. As reported by Roy et al. [Roy et al., 2009], graph-based tools can detect code clones of Type-1 to Type-4. The technique can also be applied to both source [Liu et al., 2006, Krinke, 2001] and compiled code [Chae et al., 2013]. Nevertheless, it suffers from high computational complexity like the tree-based approaches, and requires add-on optimisations to feasibly work in practice.

2.4.6 Compile Code-based Approaches

Recently, the horizon of program similarity detection has expanded from source code to compiled code which allows a detection process to be purely performed on executable files. It is specifically beneficial when the source code is absent. In the past few years, there are several studies regarding detecting cloned and plagiarised programs or mobile applications based on their compiled code files, i.e., Java bytecode or C binary code, including Chae et al. [2013], Chen et al. [2014], Gibler et al. [2013], Crussell et al. [2013], Tian et al. [2014], Tamada et al. [2004], Myles and Collberg [2004], Lim et al. [2009], Zhang et al. [2012], McMillan et al. [2012], Luo et al. [2014], Zhang et al. [2014] and [Crussell et al., 2012].

A number of the compiled code-based tools for program theft detection are based on software watermarks, i.e., a piece of value intentionally planted in a program for an identification purpose. The watermark can be created in both static and dynamic fashion. Collberg et al. [2004] introduce a software watermarking technique using a dynamic path-based approach. A watermark is implanted into a program in its runtime branch structure. The method is robust against several attacks and can be applied to either Java bytecode or native Intel IA-32 code. However, the method adds overheads to the program and slows it down by some degrees. Software birthmark is later introduced as a replacement for software watermarking. Instead of embedding a special value into a program used in watermarking technique, software birthmark aims to discover *inherent* characteristics of a program to identify its originality. So, no change or overhead embedded information has to be made to the software at all. Software birthmark can be extracted in either a static or dynamic manner. Lim et al. [2009] detect Java program theft using software birthmarks created from control flow information of software executables. Their method extracts a *flow path* which is a sequence of instructions obtained from a control flow graph of a program. They detect similar behaviours between two programs by using semi-global alignment to match flow paths of any two programs.

Besides software watermark and birthmark, directed acyclic graph (DAG) is selected by Luo et al. [2014] to build a resilient detector for obfuscated code called

CoP. The tool applies semantic similarity measure with some levels of fuzziness to overcome obfuscations. Each binary program is converted into a DAG. Linearly independent paths are created from these DAGs using depth-first search. A linearly independent path is chosen from a plaintiff program to find a match of semantically equivalent blocks in DAGs of suspicious programs using LCS. Many optimisations are applied to the LCS algorithm to deal with obfuscation. The experiment shows that CoP has higher accuracy compared to other source-based similarity tools (MOSS, JPLag), and binary-based tools (Bdiff², and DarunGrim³).

Hemel et al. [2011] create Binary Analysis Tool (BAT) to find software license violations in binaries code. The tool deploys several techniques to detect clones between software binaries including matching of string literals, compressed files similarity using normalised compression distance (NCD), and binary deltas. The tool's evaluation on the ground truth of ten known binaries shows considerably high precision and recall.

Lastly, Zhang et al. [2014] propose a method, claimed to be the first, to discover plagiarism at the algorithm level. It is a hybrid method requiring source code of a plaintiff program and binary code of a suspected one. The main idea to locate a signature of the program, i.e., *core values*. The core values capture crucial runtime values inherently related to that program. The similarity computation phase applies LCS over two core value sequences.

The approach of analysing compiled code for program similarity detection has shown to be a promising solution. It has a benefit to commercial software plagiarism detection with absence of the source code of the suspected program. Importantly, compiled executables mostly remove all formatting differences (Type-1), automatically normalise variables (Type-2), and mainly capture the semantics of the programs. Thus, they are supposed to detect all types of clone, especially Type-3 and Type-4 clones. Moreover, the evaluation shows that it outperforms source-based techniques [Luo et al., 2014].

Besides detection, there are studies that try to enhance the performance of

²Bdiff tool: <http://sourceforge.net/projects/bdiff/>

³DarunGrim tool: <http://www.darungrim.org/>

existing tools by looking for more clones from the compiled version of the code such as Jimple code [Selim et al., 2010], bytecode [Chen et al., 2014, Kononenko et al., 2014], or assembler code [Davis and Godfrey, 2010]. Using these compiler-based intermediate representation for clone detection gives satisfying results mainly by increasing recall of the tools. We also investigate the use of compilation and decompilation as a method to enhance code clone detection in this thesis (see Chapter 6).

2.4.7 Model-based Approaches

Models, which are used extensively in the design and development of embedded systems, can be cloned as well. Due to their inherent differences from source code, a dedicated technique is required to detect duplications in models. Deissenboeck et al. [2008] is among the first to present a clone detection approach for MATLAB/Simulink/TargetLink models. Their approach is based on subgraph similarity with a heuristic to reduce the number of pairwise comparisons, and a model splitting method to increase scalability. Later, Pham et al. [2009] develop a clone detector for MATLAB/Simulink models called *ModelCD*. The tool consists of two techniques to detect model clones: eScan (for exactly-matched clones) and aScan (for approximately-matched clones). The eScan uses canonical labeling as an optimisation technique for graph isomorphism computation. For aScan, an approximate matching of graphs is done using Exas, a vector-based representation and feature extraction method [Nguyen et al., 2009a]. The clones are grouped using locality-sensitive hashing (LSH) [Slaney and Casey, 2008]. Pairwise comparisons are performed only between items within the same group. A comparison of ModelCD to Deissenboeck's approach shows that both tools offer 100% precision on exactly-matched clones, while ModelCD reports additional clone pairs and clone groups.

Alalfi et al. [2012] target near-miss clones in Simulink models. They create SIMONE, a near-miss model clone detector, based on the foundation of the NiCad source-based clone detection tool. SIMONE relies on the Simulink TXL grammar to parse Simulink model files which contain the textual serialisation of the models.

The tool sorts and renames model components (e.g., blocks, lines, ports, branches) to detect near-miss model clones. A comparison of SIMONE to ConQAT, a well-known open source clone detection tool created by CQSE⁴, on three models show that SIMONE can find some challenging Type-3 clone pairs that are missed by ConQAT.

2.4.8 Other Approaches

Besides the previously presented six approaches, there are techniques that adopt from other research areas such as information theory, information retrieval, data mining, and machine learning. These techniques show promising results and open more possibilities in code similarity measurement. We discuss some of the work here.

Davies et al. [2013] introduce a method called *Software Bertillonage* to find matches between software archives in either binary or source form. The authors apply Bertillonage method, i.e., a biometric-based forensic analysis technique to identify a person used in the 19th-century France, to software. The method generates quick, easy, and efficient software fingerprints (or signatures) for similarity comparisons. The authors rely on *anchored class signatures* generated from subject class files or source files to represent a program. Each anchored class signature consists of tokens of class names, method names, and field signatures. These signatures are compared using the Jaccard coefficient, inclusion, and containment similarity. The results from an experiment confirm that Bertillonage method is effective and scalable in locating similar code between different software archives within Maven repository.

Chen et al. [2004] introduce a new way of plagiarism detection using *Kolmogorov complexity* [Li and Vitányi, 2008]. They create a tool called *Software Integrity Diagnosis (SID)* system. The authors invent a *TokenCompress* compression algorithm to reduce the size of duplicated code before doing the modified version of *Lempel-Ziv (LZ)* data compression. The authors approximate the distance $d(x, y)$ between two programs (x and y) using Kolmogorov complexity by

⁴<https://www.cqse.eu/en/products/conqat/overview>

$$d(x,y) \approx 1 - \frac{Comp(x) - Comp(x|y)}{Comp(xy)} \quad (2.1)$$

where $Comp(x)$ represents a compressed version of program x . An experiment shows that SID offers the same performance as MOSS and JPlag. Moreover, SID produces better results in a case of code insertion and boiler-plate code.

McMillan et al. [2012] choose Latent Semantic Indexing (LSI) techniques mainly used in information retrieval to find similar Java software applications. Similarly, Cosma and Joy [2012] create a tool called *PlaGate* [Cosma, 2008] for source code plagiarism using Latent Semantic Analysis (LSA). The two major benefits of LSA is being language agnostic and its independence from a parser.

Data mining technique is adapted to clone detection by Li et al. [2006]. The tool called *CP-Miner* is a hybrid tool with several optimisations. It converts a program into tokens to avoid literal changes. Then *Closed Sequential Pattern Matching (CloSpan)*, a data mining algorithm based on frequent subsequence mining, is applied to the token sequences. It includes multiple pruning methods as a post-process to remove false positives caused by too-small clone fragments, overlapped fragments, or clones with a large gap.

Chen et al. [2014] propose an efficient and scalable technique to detect cloned apps across different Android markets based on *centroid*, a geometric characteristic of a program. A 3D-CFG and its centroid are derived from an application bytecode. A similarity between two applications is derived⁸ from a distance of the two centroids extracted from the applications. The technique is found to be very fast and accurate with a very low false positive and false negative rate for cloned application detection. Moreover, it is highly scalable since the number of pairwise comparisons is decreased to only a few top results (eight as is chosen by the authors).

White et al. [2016] and Li et al. [2017] apply deep learning techniques to clone detection. They both create a probabilistic model by training on a corpus of labelled clone data. The trained model can classify if two Java code fragments are clones. White et al.'s technique is based on tree structure, called *olive trees*, while Li's

technique is based on token frequency. Both techniques have shown to perform well in an evaluation compared to the state-of-the-art clone detectors.

2.5 Benchmarks for Comparing Code Similarity Tools

Although there are a large number of clone detectors, plagiarism detectors, and code similarity detectors invented in the research community, there are relatively few studies that compare and evaluate their performances.

Burd and Bailey [2002] compared five clone detectors, CCFinder, CloneDR, Covet, JPlag, and Moss, for preventive maintenance tasks. Bellon et al. [2007] presented and used a framework for comparing and evaluating six clone detectors: Dup, CloneDr, CCFinder, Duplix, CLAN, and Duploc. The clone oracle is created by Bellon, one of the authors, by manually looking at 2% of the merged clone pairs from the six participating tools. The Bellon's framework has later been used in several studies for evaluating code clone detection tools (e.g., Wang et al. [2013b], Svajlenko and Roy [2014], Koschke et al. [2006]).

Later, Roy et al. [2009] performed a thorough evaluation of clone detection tools and techniques covering a wider range of tools. However, they compare the tools and techniques using the evaluation results obtained from the tools' published papers without an experiment. In the same year, Roy and Cordy [2009a] created a mutation/injection-based automatic framework for evaluating code clone detection tools by applying mutation operators to create clones. The framework imitates code changes made to clones of Type-1 to Type-3. Hage et al. [2010] compare five plagiarism detectors in term of their features and performances against 17 code modifications. Biegel et al. [2011] compare three code similarity measures to identify code that need refactoring. Svajlenko and Roy [2014] compared recall of eleven clone detectors based on four different clone benchmarks including the Bellon's Framework, their modified version of Bellon's Framework, another extension of Bellon's Framework [Murakami et al., 2014], and their Mutation and Injection Framework.

Svajlenko et al. [2014b] build *BigCloneBench*, possibly the largest clone benchmark available to date. The benchmark is created from IJaDataset 2.0 [ASE group, 2018] of 25,000 Java systems. It contains 2.9 million files with 8 million manually validated clone pairs of Type-1 up to Type-4. Its manually-confirmed clone oracle is created by searching for methods containing keywords and source code patterns of 43 functionalities. Later, Svajlenko and Roy [2016] develop a clone evaluation framework, called BigCloneEval, that automatically measures clone detectors' recall on the BigCloneBench data set.

2.6 Scalable Code Similarity Measurement and Code Search Techniques

Scalable code similarity detector is vital in the era of Internet-driven software development. With the rise of the Internet, the amount of source code freely available online increases exponentially. This phenomenon intensifies code cloning, software plagiarism, and software license violations since source code can be easily accessed on the web. Scalable code similarity detection methods are required to tackle this challenge of ever-growing online source code data. We pick some of the new and interesting scalable tools from the literature to discuss their strengths and weaknesses here.

2.6.1 Scalable Clone Detection

Hummel et al. [2010] is among the first to present clone detection tool for Type-1 and Type-2 clones that is incremental and scalable using index-based techniques. A clone index is created from source code sequence hashes. The tool evaluation shows that it returns clones for a file in 42M SLOC Eclipse code base within 1 second. The tool can be distributed to gain even higher scalability. Lavoie et al. [2010] propose a new version of a dynamic programming algorithm called *DP-matching* and use it for clone fragment similarity calculation on a graphic processing unit (GPU). However, the evaluation results show that their GPU-based approach only slightly increases the performance of DP-matching from its CPU-based approach. Livieri

et al. [2010] present a scalable approach for clone detection using n -gram matching. Their evaluation of a tool implementing the idea, called *Yocca*, shows that it is more scalable than CCFinder and Simian. However, the authors only discuss scalability and did not report the clone detection accuracy of the tool.

Inoue et al. [2012] propose a system called *Ichi Tracker* that leverages the power of three code search engines: Google Code Search, Koders⁵, and SPARS/R⁶. The system is designed for tracking an origin and evolution of source code. Nevertheless, the Google Code Search and Koders are no longer available, which severely affects the usability of the system. Koschke [2014] presented a scalable inter-system clone detection using a suffix-tree-based algorithm. The author evaluated the use of index-based hashes of n -gram tokens to speed up the clone detection process and concluded that building an index was worthwhile only if it is reused multiple times. Moreover, he showed that software metrics and a learned decision tree increase the clone detection’s precision. Ohmann and Rahal [2014] propose an approach, called *Program It Yourself (PIY)*, for efficient source code plagiarism detection using parallel execution and clustering algorithms. PIY relies on n -grams to create document vectors and compare them using Manhattan and cosine distance metrics. Its efficiency in large-scale data is dramatically enhanced by including parallel execution and clustering methods. However, the biggest dataset tested with PIY contains approximately 23,000 files which is still relatively small compared to the current large-scale source data that can exceed millions or hundred millions of source code files.

Tamersoy et al. [2014] show an efficient approach for large-scale malware detection based on association graphs. The authors propose a method to estimate machine and program co-occurrence strength using MinHashing algorithm [Rajaraman and Ullman, 2011] and locality-sensitive hashing (LSH) [Slaney and Casey, 2008] and implement a tool called *AESOP*. The study analyses large amount of data from the Symantec Norton’s Community Watch containing 11 million machines and 43 million files. Nonetheless, the approach needs source code data that

⁵<http://code.openhub.net>

⁶<http://sel.ist.osaka-u.ac.jp/SPARS/index.html.en>

contain associations between the code and their owners, which do not always exist. Keivanloo et al. [2014] presents a code search system aiming to find working code examples. It tackles the problem of current code search systems that rely on API names as search keywords by proposing *abstract programming solution* extraction approach. Its evaluation on IJaDataset 2.0 shows that the approach outperforms an industrial Ohloh Code search engine on finding working code examples. However, the query set in the evaluation is limited to only 15 queries, and the comparison of the two systems is performed on a different data set, which makes the findings not generalised. Svajlenko et al. [2014b] present a large-scale clone detection solution by utilising classic clone detectors. The authors introduce a scalable non-deterministic algorithm called *shuffling framework*. The framework partitions the dataset into small subsets that fit with the tool’s input size and environments. The experimental results show that the framework can enable Simian and NiCad to execute against large datasets. However, the framework suffers from problems of clone-line mismatches, high generation time of inverted index, and a bottleneck from sequential subset generation.

Sajnani et al. [2016] create a scalable code detection tool called *SourcererCC*. The tool is a token-based detector based on an optimised inverted index to scalably retrieve clone pair candidates within a short amount of time. The authors incorporate two filtering heuristics, sub-block overlap and token position, to dramatically reduce the number of pairwise comparisons. The tool reports high recall and precision compared to several state-of-the-art clone detectors. It also scales to the IJaDataset 2.0, which contains 250M lines of Java source code. Nishi and Damevski [2018] extend SourcererCC using adaptive prefix filtering to obtain higher clone detection scalability.

Svajlenko and Roy [2017] adopted Sajnani’s sub-block heuristic into their scalable clone detector, *CloneWorks*. The tool’s scalability is enhanced using partitioning of input code fragments to look for clones that fit within an allowed memory limit. They use a slightly modified version of Jaccard similarity to detect clones. *CloneWorks* offers high precision and recall of Type-1, Type-2, and Type-3 clones

on BigCloneBench compared to iClones, NiCad, and SourcererCC. CloneWorks gives a much faster detection speed than SourcererCC on BigCloneBench. The tool finishes its clone detection in 4 hours (conservative configurations) and 10 hours (aggressive configurations) compared to 110 hours by SourcererCC.

Oreo is a scalable clone detector created by [Saini et al., 2018] that integrates deep learning, information retrieval, and software metrics. By training a deep neural network model on 24 software metrics of cloned and non-cloned pairs reported by SourcererCC from 50,000 GitHub projects, the tool is capable of detecting a large number of challenging Type-3 and Type-4 clone pairs. Oreo completes a clone detection on IJaDataset 2.0 within about a day.

2.6.2 Code Search

Internet-scale code search is an emerging field of research to find source code data on the Internet for code reuse, bug fixing, or program comprehension [Gallardo-Valencia and Sim, 2009].

There are several tools available for code search. One can use Google as a code search engine by choosing the search keywords from desired functionalities [Sim et al., 2011]. There are also dedicated code search engines such as Krugle, searchcode, Codata, or Black Duck Open Hub Code Search (formerly known as Koders) that take programming language structure into account while searching. Researchers also create code search techniques and tools for their studies and some of them are later opened for free of use.

Linstead et al. [2009] invented Sourcerer, a source code retrieval system on the Internet-scale with million lines of code. Bajracharya et al. [2010] use structural semantic indexing (SSI) to return code examples based on similarity of API usage. The evaluation of 346 jar files from the Eclipse framework shows that SSI-based search schemes are preferred over the baseline schemes which do not include usage similarity in the search. They used the tool to collect and analysed 4,632 Java projects from SourceForge and Apache. Martie et al. [2017] reflect that code search is an iterative process where information seekers need to keep adapting their search queries until they find relevant results. They present two tools, CodeLikeThis

(CLT) and CodeExchange (CE), to facilitate iterative code search and perform a user study to show that the tools could improve code search experience. Niu et al. [2017] improve the ranking schema of code results by applying a learning-to-rank machine learning algorithm. They found that the approach outperforms five existing ranking schemas based on the normalised discounted cumulative gain (NDCG) by at least 35.65%. The work by Gu et al. [2018] uses deep learning techniques called CODEnn (Code-Description Embedding Neural Network) to match code snippets and natural language descriptions in the query using joint high-dimensional embedding vectors.

We refer the readers to a book by Sim and Gallardo-Valencia [2013] which presents a comprehensive list of code search studies including the motivation and behaviours of programmers to search for code, a user study on Internet-scale code search, and the infrastructures and techniques for code and software component search engines.

2.6.2.1 Code Clone Search

In this thesis, we focus on a specific kind of code search called *code clone search*. Code clone search is a special case of code search where a piece of code is given as a query instead of natural text keywords. By executing the query, a clone search system returns a list of clones of the query. Code clone search differs from code clone detection because it is query-centric. Instead of looking for a complete set of clone pairs or clone groups in given code corpora as in clone detection, a clone search tool retrieves only clones that are associated with the query. Due to the similarity between code clone detection and clone search and the limited number of clone search tool available, sometimes clone detectors are also used to search for similar code. Here, we discuss techniques that are dedicatedly invented for clone search.

Lee et al. [2010] search for clone using structural similarity based on R*Tree indexing structure. The technique searches for clones within 492 open source projects in less than a second. *Exemplar* [Grechanik et al., 2010] leverages program analysis with information retrieval to search for highly relevant applications. The

tool searches for similar applications based on similarity of their API calls. A user study with 39 professional Java programmers showed that Exemplar outperformed the SourceForge search module in searching for relevant applications. *Portfolio* [McMillan et al., 2011] uses multiple techniques including natural language processing, PageRank, and spreading activation network to find relevant functions and projects. Keivanloo et al. presented a real-time code clone search which utilises ontologies to expand the search keywords [Keivanloo et al., 2012]. The authors also present other variations of real-time clone search system using multi-level indexing [Keivanloo et al., 2011], and abstract programming solutions [Keivanloo et al., 2014].

Ishio et al. [2017] present a scalable approach for detecting clone-and-own software packages using b-bit minwise hashing technique. Then, an aggregated file similarity is applied to rank the returned search components. The technique gives a recall score of 0.907 in the evaluation. Kim et al. [2018] create a *FaCoy* code-to-code search system that leverages the information on Stack Overflow to expand the keywords in the search query. The tool aims for searching semantically similar code. The evaluation shows that the technique can return code snippets with similar runtime behaviours to the query snippet and are useful for patch recommendations.

2.7 Chapter Summary

This chapter provides a literature survey of the related work on code similarity including code clones, software plagiarism detection, and software license violations. We also discuss the existing code similarity detection techniques and the newly emerging scalable approaches. The chapter ends with the benchmarks for comparing code similarity tools and scalable code clone search techniques.

The next chapter will present an empirical study that motivates the thesis author to invent an approach for a scalable clone search engine. It will present two online surveys of Stack Overflow users regarding the issues of outdated code and software license incompatibility, which are caused by code cloning.

Chapter 3

Awareness and Experience of Developers to Outdated and License-Violating Code on Stack Overflow: An Online Survey

The chapter discusses two problems of outdated code and software license violations caused by code cloning to and from Stack Overflow, a popular Q&A website, via two online surveys. The chapter presents the methodology used to perform the surveys and analyse the results. The findings show that the two issues occasionally occur on Stack Overflow and the survey participants suffer from them. The survey results suggest that some guidelines from Stack Overflow and/or an automated support system are needed to mitigate the problems.

3.1 Motivation

Recent research shows that outdated third-party code and software license conflicts are ramifications of code cloning. Xia et al. [2014] report that a large number of open source systems reuse outdated third-party libraries from popular open source projects. Using outdated code has detrimental effects to software since they may introduce vulnerabilities. On the other hand, German et al. [2009] found that code cloning leads to software license conflicts among different systems. Incorporating

code with incompatible license into software is also troublesome, since it may lead to legal issues.

The Internet encourages fast and easy code cloning by sharing and copying source code to and from online sources. Developers nowadays do not only clone code from local software projects, but also from online sources, especially Stack Overflow [Acar et al., 2016, Abdalkareem et al., 2017, An et al., 2017, Yang et al., 2017]. Unfortunately, a number of code snippets on Stack Overflow are found to be problematic. Acar et al. [2016] discovered that many code snippets provided as solutions on Stack Overflow are workarounds and occasionally contain defects or vulnerabilities. They performed a user study and found that although Stack Overflow helped developers to solve Android programming problems quicker, the website offered less secure code than books or the official Android documentation. Only 17% of the Stack Overflow discussion threads that the participants visited during the study contained secure code snippets. They found that a problematic code fragment copied from Stack Overflow by participants in their study also occurs in 187,291 Android apps from Google Play. In addition, An et al. [2017] investigated clones between 399 Android apps and Stack Overflow posts and found 1,226 code snippets that were reused from 68 Android apps. Importantly, they observed 1,279 cases of potential license violations from such cloning.

Asking and answering questions on Stack Overflow involves source code snippets, either in a question, an answer, or both. While many code examples are written from scratch, several of them are copied from other sources. Since Stack Overflow is a website, the code examples are rarely tested and updated as in typical software projects. Hence, the copied code snippets might not be up-to-date with their originals. Besides, some snippets are copied from software systems with stricter licenses than the Stack Overflow’s Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0).

A study in this chapter is motivated by several discussion threads about outdated answers and license of code on Stack Overflow, e.g., meta.stackexchange.com/questions/131495, meta.stackexchange.com/questions/11705,

meta.stackexchange.com/questions/12527, meta.stackexchange.com/questions/25956, and meta.stackoverflow.com/questions/321291. In these discussion threads, Stack Overflow users express their concerns about the two problems, and there is no clear solution yet. To gain insights into the problems, this chapter resorts to a study using two online surveys of Stack Overflow developers who 1) regularly answer programming questions with code snippets and 2) regularly reuse code snippets from Stack Overflow. We aim to assess *the developers' awareness and experience to outdated code and software license violations caused by code cloning to and from Stack Overflow*.

3.2 Stack Overflow: A Popular Programming Q&A Website

Stack Overflow is a popular online programming community with 7.6 million users, 14 million questions, and 23 million answers¹. It allows programmers to ask questions and give answers to programming problems. It is a gold mine for software engineering research and has been put to use in several studies. Regarding developer-assisting tools, *Seahawk* is an Eclipse plug-in that searches and recommends relevant code snippets from Stack Overflow [Ponzanelli et al., 2013]. A follow-up work, *Prompter*, by Ponzanelli et al. [2014] achieves the same goal but with improved algorithms.

The code snippets on Stack Overflow are mostly examples or solutions to programming problems. Hence, several code search systems use whole or partial data from Stack Overflow as their code search databases, e.g., Keivanloo et al. [2014], Park et al. [2014], Stolee et al. [2014], Subramanian and Holmes [2013], Diamantopoulos and Symeonidis [2015]. Furthermore, Treude and Robillard [2016] use machine learning techniques to extract insight sentences from Stack Overflow and use them to improve API documentation.

Another research area is knowledge extraction from Stack Overflow. Nasehi et al. [2012] studied what makes a good code example by analysing answers from

¹Data as of 21 August 2017 from <https://stackexchange.com/sites>

Stack Overflow. Similarly, Yang et al. [2016] report the number of reusable code snippets on Stack Overflow across various programming languages. Wang et al. [2013a] use Latent Dirichlet Allocation (LDA) topic modelling to analyse questions and answers from Stack Overflow so that they can automatically categorise new questions. There are also studies trying to understand developers' behaviours on Stack Overflow, e.g., Movshovitz-Attias et al. [2013], Bosu et al. [2013], Choetkertikul et al. [2015] and Rosen and Shihab [2016].

3.3 Terminology

In this chapter, we use the term “**answerers**” to refer to Stack Overflow users who actively answer questions, which is measured by their reputation. The answerers gain a reputation from giving a helpful answer to a question and receiving votes from other users. The reputation reflects trust they gain from other users and also the quality of their answers.

We use the term “**visitors**” to refer to developers who visit Stack Overflow when they encounter programming problems. They copy code snippet(s) in a solution that is relevant to their problems and reuse them with or without modifications.

We use the term “**(potentially) license-violating code snippets**” or “**code with (potential) license conflicts**” interchangeably to refer to Stack Overflow cloned code snippets that violate or potentially violate the original license by not including the original license statement in the cloned snippets. These code snippets are automatically covered by the Stack Overflow CC BY-SA 3.0 license instead, which may or may not conflict with their original licenses.

Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)² is a license that allows the licensed content to be freely shared and adapted. However, users of the content must give attribution to the original source by providing a link to the license and state whether changes were made to the copied content. Moreover, the derivative of the content has to also be under CC BY-SA 3.0 license. Stack Overflow applies CC BY-SA 3.0 to all content on the website³.

²<https://creativecommons.org/licenses/by-sa/3.0/>

³<https://stackoverflow.com/legal/terms-of-service/public>

3.4 Contributions

This chapter makes the following primary contributions:

1. **Awareness of Stack Overflow answerers to outdated and potentially license-violating code on Stack Overflow:** We performed an online survey and collected answers from 201 highly-ranked Stack Overflow answerers. We found that the answerers occasionally cloned code snippets from open source projects to Stack Overflow answers. While they were aware of outdated code snippets in their answers, 19% of the participants rarely or never fixed the code. 99% of the answerers never included a software license in their snippets and 69% never checked for license conflicts.
2. **Awareness of Stack Overflow visitors to outdated and potentially license-violating code on Stack Overflow** We performed another online survey of 87 Stack Overflow visitors. 66% of the Stack Overflow visitors experienced problems from reusing Stack Overflow code snippets, including outdated code. They were generally not aware of the CC BY-SA 3.0 license, and more than half of them never checked for license compatibility when reusing Stack Overflow code snippets.

3.5 Research Methodology

We followed the principles of survey research by Pfleeger and Kitchenham [2001] and Kitchenham and Pfleeger [2002] by setting a specific and measurable objective, designing and scheduling the survey, selecting participants, analysing the data, and reporting the results. We now discuss each of them in detail.

3.5.1 Survey Objective

The survey is conducted to answer the following five research questions.

1. **RQ1 (Sources of Stack Overflow Answer Snippets):** *Where are the code snippets in Stack Overflow answers from?*
2. **RQ2 (Answerers' Awareness to Outdated Code):** *Are Stack Overflow answerers aware of outdated code in their answers?*

3. **RQ3 (Answerers' Awareness to Potential License Violations):** *Are Stack Overflow answerers aware of potential software license violations caused by code snippets in their answers?*
4. **RQ4 (Visitors' Problems from Stack Overflow Code Snippets):** *What are the problems Stack Overflow visitors have experienced from reusing code snippets on Stack Overflow?*
5. **RQ5 (Visitors' Awareness to Potential License Violations):** *Are Stack Overflow visitors aware of or experienced code with license conflicts on Stack Overflow?*

3.5.2 Survey Design and Schedule

The study was conducted using unsupervised online surveys. We designed the surveys using Google Forms and created two versions of the survey: **the answerer survey** and **the visitor survey**. Both surveys were completely anonymous, and the participants could decide to leave at any time. They did not collect any sensitive personal information from the participants and were approved for an ethical waiver by the designated ethics officer in the Computer Science Department at University College London (UCL). The complete version of the two surveys can be found in Appendix A.2 and Appendix A.3.

3.5.2.1 The answerer survey

The survey contained 11 questions: 7 Likert's scale questions, 3 yes/no questions, and one open-ended question for additional comments. The first two questions were mandatory while the other 9 questions would be shown to the participants based on their previous answers. The survey collected information about the participants' software development experience, their experience of answering Stack Overflow questions, sources of the Stack Overflow snippets they used to answer questions, awareness of outdated code in their answers, their concerns regarding license when cloning code snippets to Stack Overflow, and their additional feedbacks. The survey was open for participation for 50 days, from 25th July 2017 to 12th September 2017, before we collected and analysed the responses.

Table 3.1: The Stack Overflow answerer taken the surveys

Target group	Reputation	Sent emails	Answers	Rate
Answerer Group 1	963,731–7,674	300	117	39%
Answerer Group 2	7,636–6,999	307	84	27%
Total	–	607	201	33%

3.5.2.2 The visitor survey

The survey consisted of 16 questions: 9 Likert's scale questions, 3 yes/no questions, 2 multiple-choice questions, and 2 open-ended questions. The first four questions were mandatory while the other twelve questions would be shown to the participants based on their previous answers. The survey collected information about the participants' software development experience, the importance of Stack Overflow in their opinion, their reasons for reusing Stack Overflow snippets, problems they faced from Stack Overflow code snippets, their awareness to software license of code examples on Stack Overflow, and their additional feedbacks. The survey was open for participation for two months, from 25th July 2017 to 25th September 2017, before we collected and analysed the responses.

3.5.3 Participant Selection

The participants of the answerer and the visitor survey did not overlap. We used the following methods to select the participants for our two surveys.

3.5.3.1 The answerer survey

We selected the participants for our answerer survey based on their Stack Overflow reputation. On Stack Overflow, a user's reputation reflects how much the community trusts them. A user earns reputation when he or she receives upvotes for good questions and useful answers. For example, they gain reputation when they receive an upvote for their question (+5) or their answer (+10), or when their answer is accepted (+15)⁴. Thus, the reputation score is an indicator of Stack Overflow user's skills and their involvement in asking and answering questions on the site.

The participants were invited to take the survey via email addresses publicly

⁴Stack Overflow Reputation: <https://stackoverflow.com/help/whats-reputation>

available on their Stack Overflow and GitHub profiles. We selected the answerers based on their all-time reputation ranking⁵. Then, we separated them into two groups (see Table 3.1) with roughly 300 participants in each group so that we can compare and contrast the results between them. The first group had a reputation from 963,731 (the highest ranked user) to 7,674 and the second group had a reputation from 7,636 to 6,999. We sent out 300 and 307 emails (excluding undelivered ones) to the two groups respectively.

3.5.3.2 The visitor survey

We adopted non-probability convenient sampling to invite participants for the visitor survey. To take the visitor survey, the participants must have visited Stack Overflow for solving programming tasks at least once. The participants were invited to take the survey via five channels. The first channel was via the thesis author's social media post (Facebook) inviting software developers who had experience in copying code snippets from Stack Overflow to take the survey. The second channel was a popular technology news and media community in Thailand called blognone.com which attracted a high number of Thai software developers. We posted an invitation to the visitor survey in a discussion forum mentioning the requirements to take the survey. The third channel collected answers from the University of Molise in Italy, where a colleague of the thesis author works. The last two channels are the `comp.lang.java.programmer` group and the Software Engineering Facebook page. The number of participants taken the survey is shown in Table 3.2.

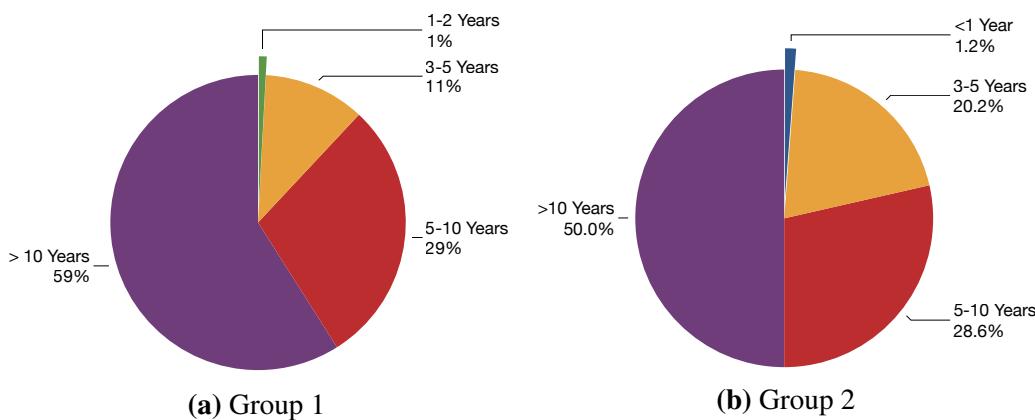
3.5.4 Data Analysis

Google Forms provide a helpful summary of responses from the online surveys, which we partially relied on when analysing the answers. In addition, we also performed a manual analysis of the results. For the visitor survey, the results were collected separately from each group. Thus, we downloaded the responses in comma-separated values (CSV) files and merged the results before the analysis.

⁵Stack Overflow Users (data as of 25th July 2017): <https://stackoverflow.com/users?tab=Reputation&filter=all>

Table 3.2: The Stack Overflow visitors taken the survey

Group	Answers
Social media (Facebook posts)	47
Blognone.com	32
University of Molise	6
comp.lang.java.programmer	3
Software Engineering Facebook page	1
Total	89

**Figure 3.1:** Experience of the Stack Overflow answerers

3.6 Results and Discussions

We collected and analysed the results after we closed the surveys on 12th and 25th September 2017. We now discuss the results from the answerer survey followed by the visitor survey.

3.6.1 The answerer survey

We received 117 answers (39% response rate) from the first group and 84 answers (27% response rate) from the second group of Stack Overflow answerers. The response rate from both groups was high considering other online surveys in software engineering [Punter et al., 2003].

3.6.1.1 General Information

The majority of users in both groups are experienced developers with more than 10 years of experience or between 5 to 10 years as depicted in Figure 3.1. There are 59% of the answerers in Group 1 and 50% of the answerers in Group 2 that have

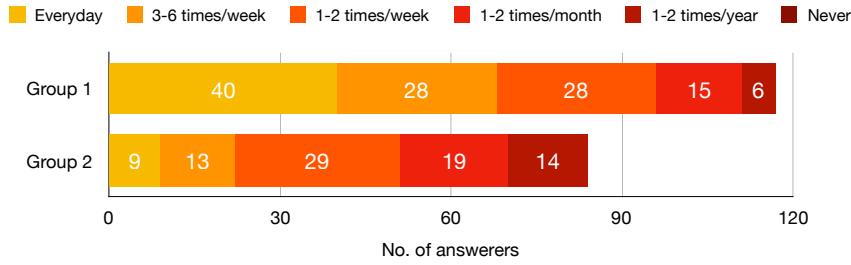


Figure 3.2: Frequency of answering questions

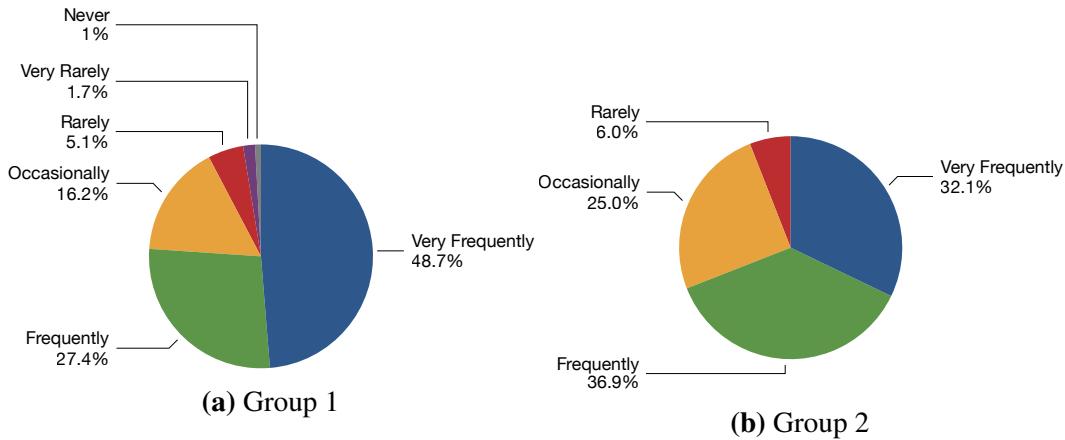


Figure 3.3: Frequency of answering questions with code snippet(s)

more than 10 years of software development experience.

The participants are active users and regularly answer questions on Stack Overflow (see Figure 3.2). Ninety-six (82%) and fifty-one (61%) of answerers from Group 1 and Group 2 answer questions at least once a week. More than half of the answerers very frequently (81–100% of the time) or frequently (61–80% of the time) include code snippets in their answers. To break down into two groups as depicted in Figure 3.3, Group 1 very frequently (48.7%) and frequently (27.4%) provide code examples when answering. Likewise, Group 2 follows the same trend (very frequently for 32.1% and frequently for 36.9%). Interestingly, there is one participant in the first group who never includes code snippet in his/her answer. Thus, the results after this question are from 116 participants of the first group.

3.6.1.2 RQ1: Sources of Stack Overflow Answer Snippets

Where are the code snippets in Stack Overflow answers from?

To answer this research question, we asked the participants for the original source of their code examples. We provided six options (allowing more than one answer) including *1) I copied them from my own personal projects, 2) I copied them from my company's projects, 3) I copied them from open source projects, 4) I wrote the new code from scratch, 5) I copied the code from the question and modified it for the answer, and 6) Others.* The answers are shown in Figure 3.4. Participants in Group 1 mainly write new code from scratch (116) or copy from the code snippets in question and modify it for the answer (112), followed by copying from their personal projects (105), open source projects (77), other sources (59), and company projects (48). For Group 2, the main source is also writing code from scratch (83), followed by copying from personal projects and modifying from the question (77), open source projects (56), other sources (40), and company projects (31). There are 133 answerers out of the total 201 from the two groups who have cloned code snippets from open source projects into their answers at least once. We are interested in this type of clones and will investigate further in the later RQs.

To answer RQ 1, we found that answering questions by writing the new code from scratch is the most popular choice for Stack Overflow answerers followed by modifying the code in question or copying from personal projects. Other less popular choices include copying code from open source projects and other sources. Copying code from company projects is the least popular choice.

3.6.1.3 RQ2: Answerers' Awareness to Outdated Code

Are Stack Overflow answerers aware of outdated code in their answers?

Half of the top answerers on Stack Overflow are aware of outdated code in their answers. Seventy-one participants (61.2%) of Stack Overflow answerers in Group 1 have been notified of outdated code in at least one of their answers. The ratio drops to forty participants (47.1%) in Group 2. We asked a follow-up question regarding

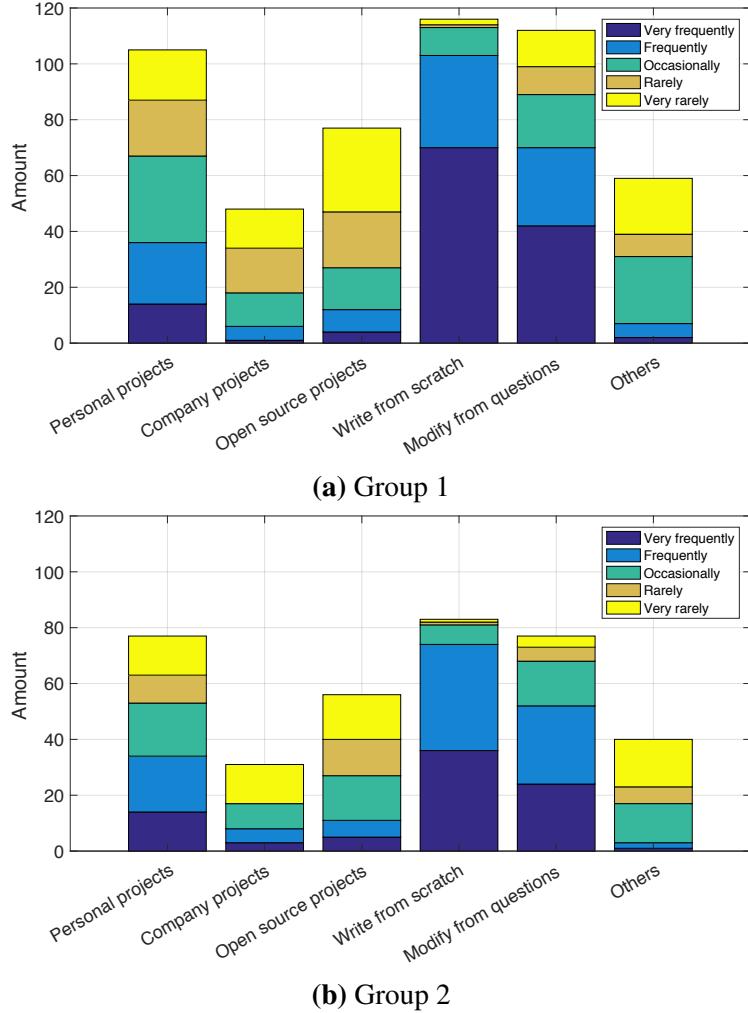


Figure 3.4: Sources of code snippets in Stack Overflow answers

the frequency of being notified of outdated code in their answers. We found that only 0.9% and 5.2% of the answerers in Group 1 have frequently or occasionally been notified. The answerers in Group 2 have very frequently and occasionally been notified for 2.4% and 3.5% respectively. Please note that we found inconsistencies between the answers to these two questions. The percentage of participants who have “Never” been notified of outdated code in their Stack Overflow answers are 38.8% and 52.9% for Group 1 and Group 2 respectively. However, the answers for the frequency of being notified equal to “Never” decrease to 28.4% and 43.5% for Group 1 and 2 respectively (see Figure 3.5 and Figure 3.6).

We then asked the participants who have been notified of their outdated code (83 and 48 participants from Group 1 and 2 respectively) a follow-up question

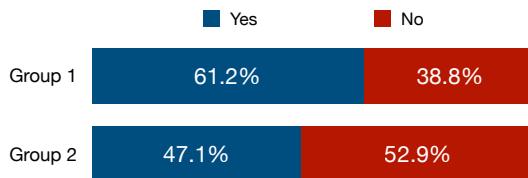


Figure 3.5: Percentage of answerers who are notified of outdated code in their Stack Overflow answers.

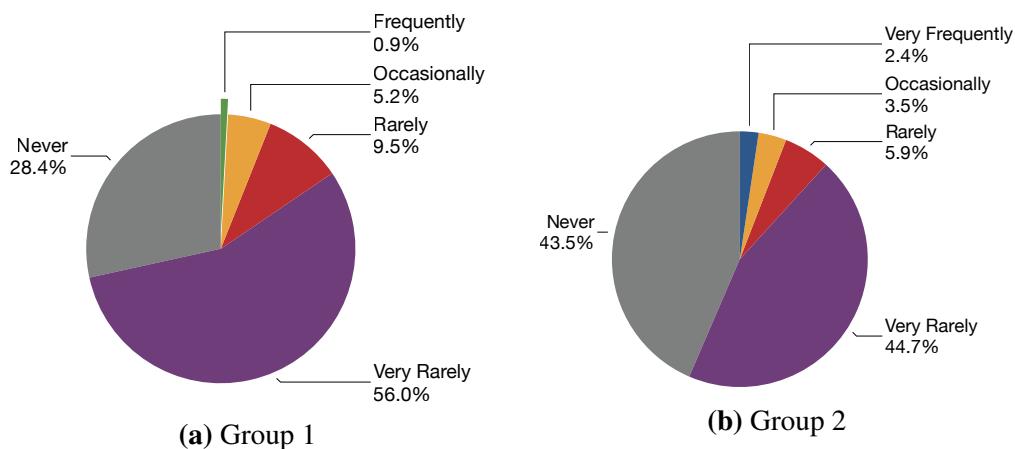


Figure 3.6: Frequency of being notified of outdated code in the answerers' answers.

“how frequently did you fix your outdated code on Stack Overflow?” The answers, depicted in Figure 3.7, show that more than half of them frequently fix the outdated code snippets. However, there are 17 (19.8%) and 9 (18.8%) participants in Groups 1 and 2 who rarely, very rarely, or never fix their code.

Regarding the issue of outdated code, one participant expresses their concern in the open comment question:

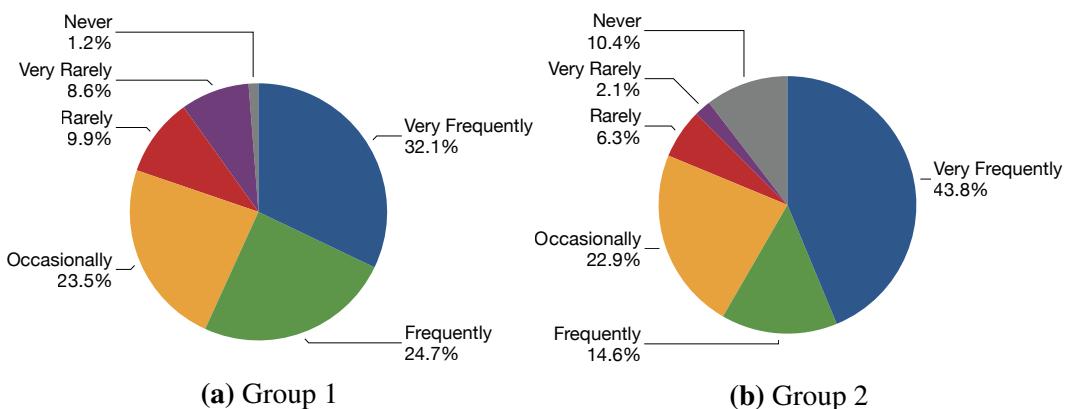


Figure 3.7: Frequency of the answerers fixing their outdated code.

The main problem for me/us is outdated code, esp. as old answers have high Google rank so that is what people see first, then try and fail. Thats why we're moving more and more of those examples to knowledge base and docs and rather link to those.

On the other hand, another participant does not worry about his/her outdated code and how he/she handles them:

On the matter of deprecation, I almost entirely use .NET which has got different versions of the framework. Therefore, code deprecation is not often a problem since what is deprecated on one version of the framework may be the only way of solving a given problem on an older version of the framework. I may also have to add that questions I tend to answer are about how to solve general coding problems so they are not usually subject to deprecation.

To answer RQ 2, Stack Overflow answerers are aware of outdated code in their answers. Nonetheless, there are approximately 19% of the answerers who rarely or never fix their outdated code for which they have been notified.

3.6.1.4 RQ3: Answerers' Awareness to Potential License Violations

Are Stack Overflow answerers aware of potential software license violations caused by code snippets in their answers?

As shown in Figure 3.8, more than half of the answerers in both groups, 72 (62.1%) and 53 (62.3%) respectively, are aware that Stack Overflow apply CC BY-SA 3.0 to content in the posts, including code snippets, while the rest of 44 (37.9%) and 32 (37.6%) are not.

Almost every answerer in both groups, 114 out of 116 (98%) and 84 out of 85 (99%) respectively, do not include license statement in their code snippets (as shown in Figure 3.9). Some of the participants explained the reason in the open comment question which we summarised into three groups as follows. First, they choose to post only their own code or code that is adapted from the question. The

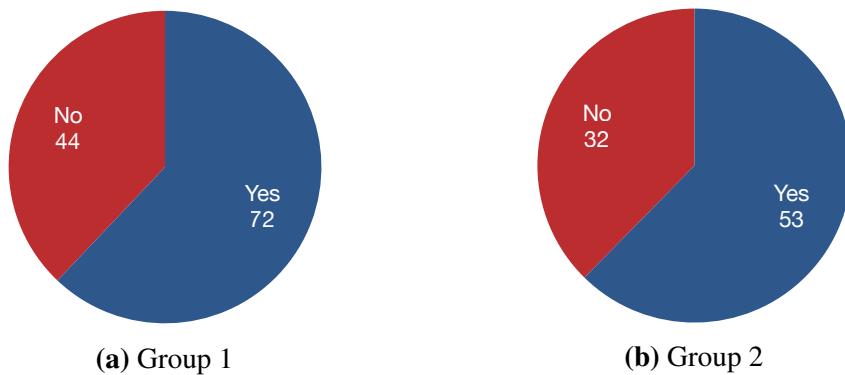


Figure 3.8: Awareness of answerers to Stack Overflow CC BY-SA 3.0 license

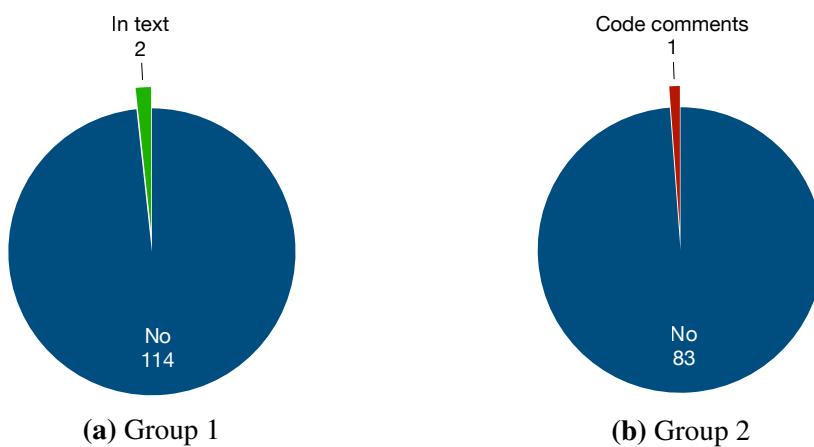


Figure 3.9: Software license in Stack Overflow code snippets.

code is then automatically subjected to Stack Overflow's CC BY-SA 3.0 without any explicit licensing statement. Second, they copy the code from their company, or open source projects that they know are permitted to be publicly distributed. Hence, no license statement is required. Third, some answerers believe that code snippets in their answers are too small to claim any intellectual property and fall under the Fair Use concept, i.e., a copy of copyrighted content that is for a limited or transformative purpose and will not be considered an infringement⁶.

While almost nobody explicitly includes a software license in their snippets, many participants include a statement on their profile page that all their answers are under a certain license. For example,

⁶The concept of fair use is enforced in the United States. Nonetheless, not every country in the world has this concept, and some countries have their own interpretation of fair use.

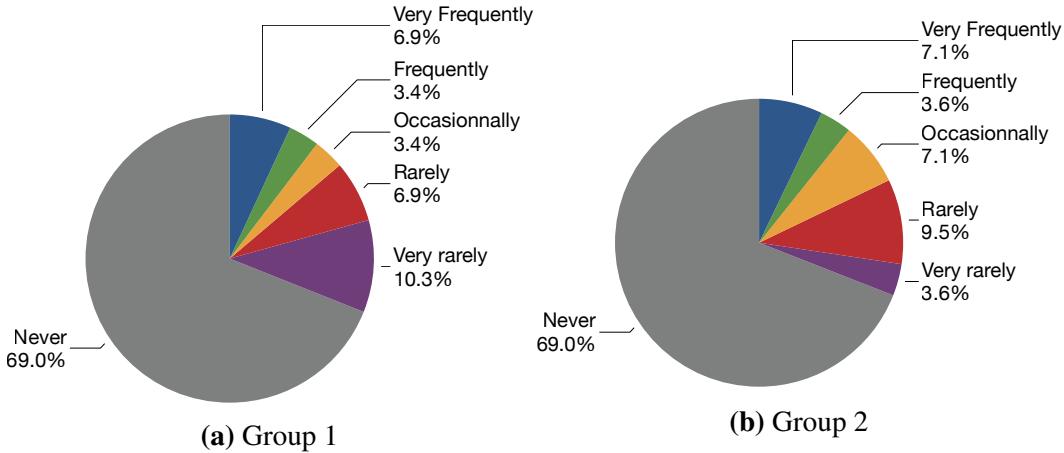


Figure 3.10: Frequency of the answerers checking license of their code snippets against Stack Overflow’s CC BY-SA 3.0.

All code posted by me on Stack Overflow should be considered public domain without copyright. For countries where public domain is not applicable, I hereby grant everyone the right to modify, use and redistribute any code posted by me on Stack Overflow for any purpose. It is provided “as-is” without warranty of any kind.

Many participants either declare their snippets to be public domain, or they grant additional licenses, e.g., Apache 2.0 or MIT/Expat.

We asked the answerers a follow-up question of how frequently they checked for conflicts between software license of the code snippets they copied to their answers and Stack Overflow’s CC BY-SA 3.0. As shown in Figure 3.10, 80 (69%) and 58 (69%) answerers from Group 1 and Group 2 did not perform the checking. There are only approximately 10% of the answerers who frequently check for license conflicts when they copy code snippets to Stack Overflow.

To answer RQ3, approximately 62% of our participants are aware of CC BY-SA 3.0 license enforced by Stack Overflow. However, 98 to 99 percent of the answerers never include software license in their Stack Overflow snippets. Sixty-nine percent never check for potential license conflicts when they copy code snippets to Stack Overflow answers.

3.6.1.5 Open Comments

To acquire additional insights, we invited every answerer for additional comments regarding their concerns of answering Stack Overflow (SO) with code snippets. Some interesting comments are selected and discussed below. The full set of answers can be found in Appendix A.1.

- Comment 1: The answerer addresses a concern of programmers reusing his/her code snippets without understanding them. Moreover, he or she discusses problems from low-quality snippets or outdated code containing security issues on Stack Overflow.

The real issue is less about the amount the code snippets on SO than it is about the staggeringly high number of software “professionals” that mindlessly use them without understanding what they’re copying, and the only slightly less high number of would-be professionals that post snippets with built-in security issues. A related topic is beginners who post (at times dangerously) misleading tutorials online on topics they actually know very little about. Think PHP/MySQL tutorials written 10+ years after mysql_ functions were obsolete, or the recent regex tutorial that got posted the other day on HackerNew (<https://news.ycombinator.com/item?id=14846506>). They’re also full of toxic code snippets.*

- Comment 2: The answerer suggests that a guidance from Stack Overflow regarding software license of code snippets will be beneficial.

When I copy code it’s usually short enough to be considered “fair use” but I am not a lawyer or copyright expert so some guidance from SO would be helpful. I’d also like the ability to flag/review questions that violate these guidelines.

- Comment 3: Similar to comment 1, the answerer addresses a concern of reusing Stack Overflow code snippets without understanding.

My only concern, albeit minor, is that I know people blindly copy my code without even understanding what the code does.

- Comment 4: As shown in RQ2, the answerer discusses a problem from outdated Stack Overflow code snippets and his/her solution.

The main problem for me/us is outdated code, esp. as old answers have high Google rank so that is what people see first, then try and fail. Thats why we're moving more and more of those examples to knowledge base and docs and rather link to those.

- Comment 5: The answerers gives insights into the quality of the Stack Overflow code snippets.

Lot of the answers are from hobbyist so the quality is poor. Usually they are hacks or workarounds (even MY best answer on SO is a workaround).

3.6.2 The visitor survey

To answer RQ4 and RQ5, we used another online survey, the visitor survey, to ask Stack Overflow visitors about their experiences of outdated code and their awareness to software license of Stack Overflow code snippets. We received 89 answers from 5 groups of Stack Overflow visitors. We combined the results and presented them in a single group as shown below.

3.6.2.1 General Information

As illustrated in Figure 3.11, 24 and 21 participants (27% and 24% respectively) from the Stack Overflow visitor survey have over 10 years and 5–10 years of experience respectively. There are 19 participants (21%) who have 3–5 years, 18 (20%) who have 1–2 years, and 7 (8%) participants who have less than a year of programming experience.

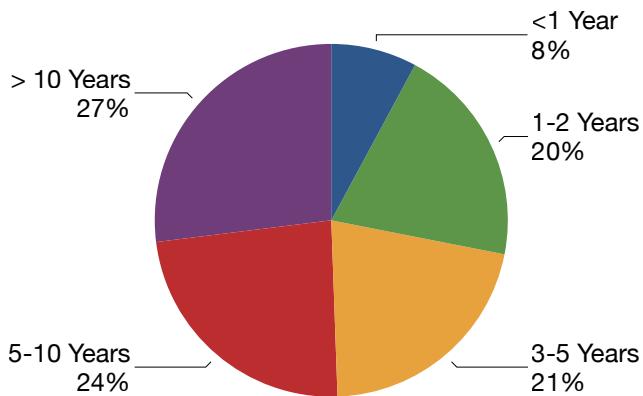


Figure 3.11: Experience of the Stack Overflow visitors

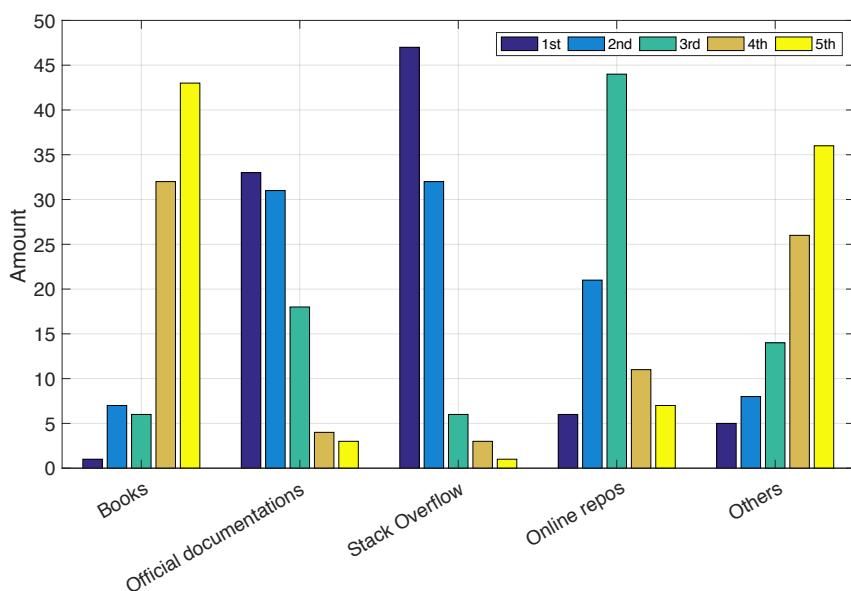


Figure 3.12: Rankings of the resources developers use to solve programming problems

3.6.2.2 Where do the developers search for programming solutions?

We asked the participants to rank five options, without a tie, that they will choose to find programming solutions. The given five options include books, official documentation, Stack Overflow, online repositories (e.g., GitHub), and others. The results are displayed in Figure 3.12. We found that 47 out of 89 participants rank Stack Overflow as the 1st option to search for programming solutions, followed by official documentation (33), online repositories (6), other resources (5), and books (1).

Since Stack Overflow is among the first resources the visitors rely on to solve programming tasks, we asked the participants how frequently they reuse

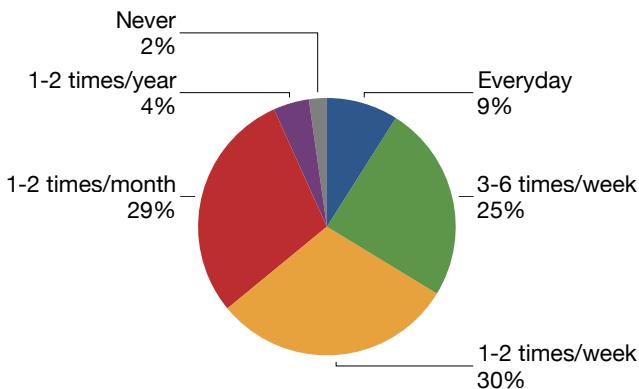


Figure 3.13: Frequency of copying code from Stack Overflow

code snippets from answers on Stack Overflow. According to the results (see Figure 3.13), we found that 57 (64%) participants actively reusing code snippets from Stack Overflow. Eight participants (9%) copy Stack Overflow code every day, 22 (25%) do copying 3-6 times a week, 27 (30%) do copying once or twice a week. There are 2 participants (2%) who never copy the code from Stack Overflow. Thus, the results after this question are from the 87 participants who used to copy code from Stack Overflow.

To understand why the participants choose to copy code snippets from Stack Overflow, we asked them to rate four reasons in Likert's scale (Strongly agree, Agree, Undecided, Disagree, Strongly disagree). The four reasons include *1) They are easy to find by searching the web, 2) They solve problems similar to my problems with minimal changes, 3) The context of questions and answers helped me understand the code snippets better, 4) The voting mechanism and accepted answers helped to filter good code from bad code*. The answers are depicted in Figure 3.14. More than 80% of the participants agree with all the four reasons. We observed only two “Disagree” and zero “Strongly disagree” answer for “Helpful context”, the lowest disagreement among the four reasons. This means most of them agree that the context of questions and answers on Stack Overflow help them understand the code snippet better.

To sum up, Stack Overflow is ranked higher than official documentation, online repositories, and books as the resource to look for programming solutions. Developers rely on Stack Overflow answers because they are easy to search for

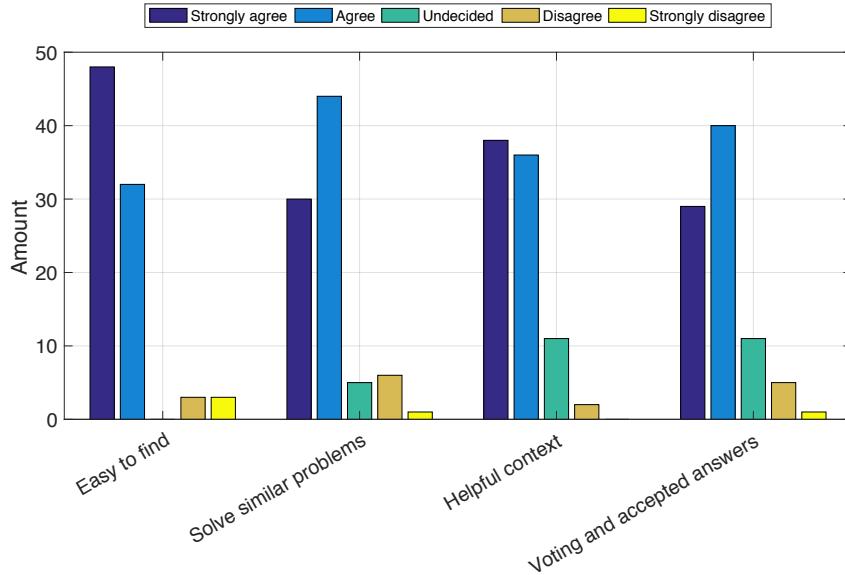


Figure 3.14: Why did you copy and reuse code snippets from Stack Overflow?

on the web. Moreover, 64% of the participants reuse code snippets from Stack Overflow at least once a week. They copy code snippets from Stack Overflow because the snippets can be found easily from a search engine, solve similar problems to their problems, provide helpful context, and offer voting mechanism and marking answers as accepted.

3.6.2.3 RQ4: Visitors’ Problems from Stack Overflow Code Snippets

What are the problems Stack Overflow visitors experienced from reusing code snippets on Stack Overflow?

We asked the visitors whether they have had any problem with reusing Stack Overflow code snippets and how often did the problems occur. Fifty-seven out of eighty-seven participants (66%) experienced a problem from reusing Stack Overflow snippets (see Figure 3.15). Among the fifty-seven participants, two participants found problems in more than 80% of the reused code snippets. Eight and sixteen faced problems from at least sixty and forty percent of the reused snippets.

The problems from reusing Stack Overflow code snippets (illustrated in

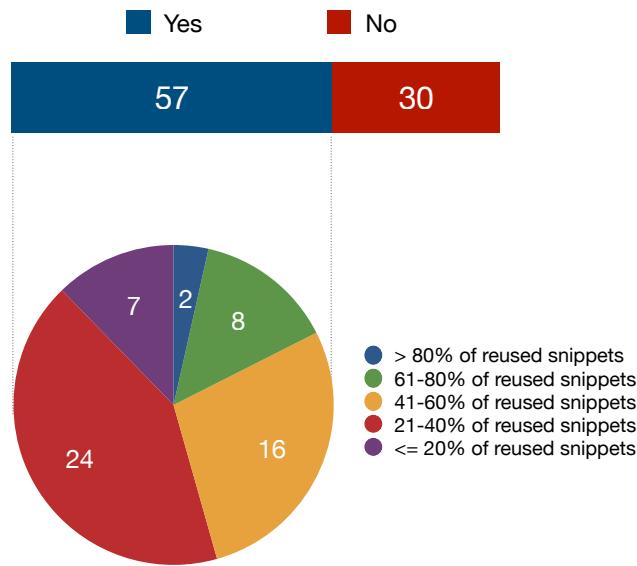


Figure 3.15: Number of visitors who experienced a problem from reusing Stack Overflow code snippets and the frequency.

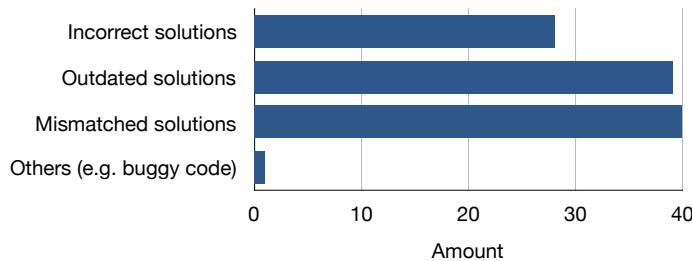


Figure 3.16: Problems from reusing Stack Overflow code snippets

Figure 3.16) include incorrect solutions, i.e., the code claims to solve the problem in the question while it does not (28 out of 57 \approx 49% participants reported this); outdated solutions, i.e., the code may work with the older versions of the library or API, but not the one they are using (39 out of 57 \approx 68% participants reported this); mismatched solutions, i.e., the code solves the problem in the question, but it is not exactly the right solution for their problem (40 out of 57 \approx 70% participants reported this); and buggy code (1 out of 57 \approx 2% participants reported this).

Stack Overflow visitors rarely report the problems back to the discussion threads (as can be seen in Figure 3.17). Among the 57 participants who encounter problems from Stack Overflow snippets, 36 of them (63.2%) never report the problems. Fourteen participants who reported the problems did so by writing a comment (10), down-voting the answer (8), contacting the answerer (2), and posting

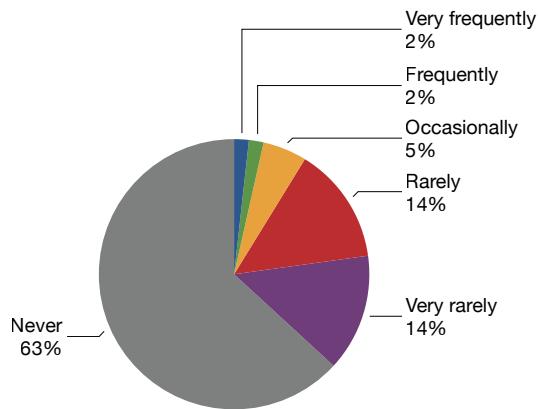


Figure 3.17: Frequency of Stack Overflow visitors reporting the problems back to Stack Overflow discussion threads

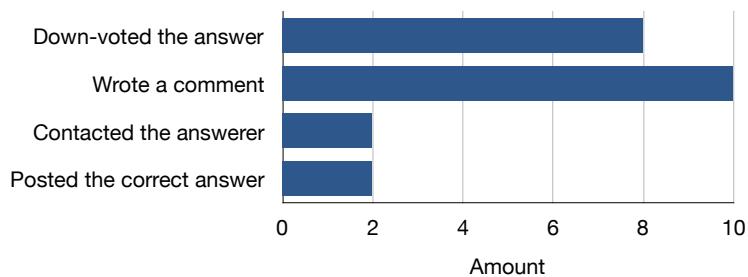


Figure 3.18: Options that Stack Overflow visitors choose to report the problems from Stack Overflow snippets

the new and correct answer (2) as summarised in Figure 3.18.

To answer RQ4, our survey results show that 57 out of 87 Stack Overflow visitors encountered a problem with reusing Stack Overflow code snippets. Ten participants experienced issues from more than 80% of the copied snippets, and sixteen participants faced problems for 40–60% of the reused snippets. The problems ranked by frequency include mismatched solutions (40), outdated solutions (39), incorrect solutions (28), and buggy code (1). Sixty-three percent of the participants never report the problems back to Stack Overflow.

3.6.2.4 RQ5: Visitors' Awareness to Potential License Violations

Are Stack Overflow visitors aware of or experienced code with license conflicts on Stack Overflow?

As depicted in Figure 3.19, 74 out of 87 (85%) Stack Overflow visitors are not aware, at the time of copying the code, that Stack Overflow applies CC BY-SA 3.0



Figure 3.19: Awareness of Stack Overflow visitor to CC BY-SA 3.0 license

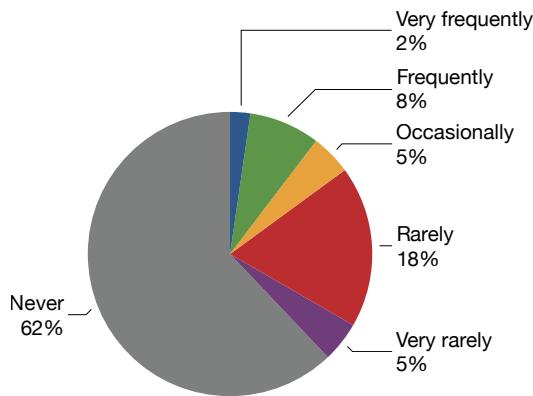


Figure 3.20: Attributions to Stack Overflow when reusing code snippets

to content in the posts, including code snippets. As a consequence, 62% of the visitors never give an attribution, which is required by CC BY-SA 3.0, to a Stack Overflow post they copied the code from (the complete statistics can be found from Figure 3.20).

Sixty-nine Stack Overflow visitors (79%) who adopted code from Stack Overflow never check if the code snippet originated from a different source (e.g., an open source project) with an incompatible license to their projects (see Figure 3.21).

Fifty-seven participants (66%) never check for potential license conflicts at all when reusing Stack Overflow code (see Figure 3.22). Lastly, 9% of the participants experienced legal issues by reusing code snippets on Stack Overflow (see Figure 3.23). We did not expect that any participant encountered legal issues as we are not aware of such cases being reported in the literature. It would be interesting to follow up on the kind of legal issues that have been encountered, however, as we designed the survey to be anonymous, it was not possible to contact the participants for further details.

To answer RQ5, 85% of the participants are not aware of Stack Overflow CC BY-SA 3.0 license, and 62% never give attributions to the Stack Overflow posts from which they copied the code snippets. We found that 66% of the visitors never



Figure 3.21: Checking for the original license of Stack Overflow code snippets

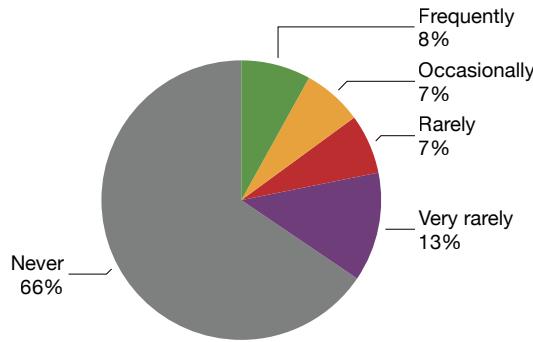


Figure 3.22: Checking for license conflicts from reusing Stack Overflow snippets

check for potential software license conflicts between Stack Overflow code snippets and their projects. Nine percent of the participants encountered legal issues.

3.6.3 Overall Discussion

By separating the answerers into two groups according to their reputation, we observed some similarities and differences in their responses. The sources of code snippets in Stack Overflow answers are similar in both groups. The answerers mainly write the code snippets from scratch aiming to answer the question. The frequencies of copying from each source, i.e., personal projects, company projects,

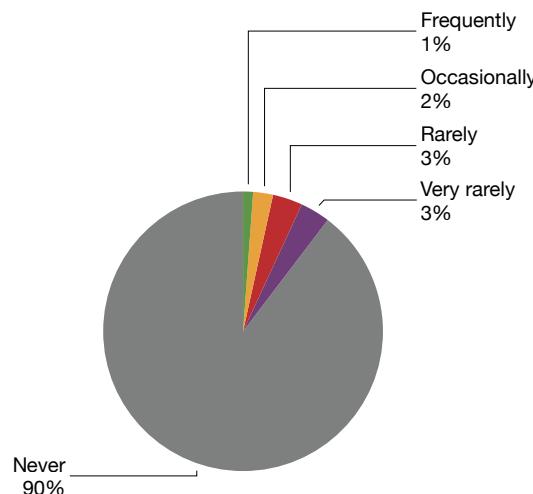


Figure 3.23: Legal issues found from reusing Stack Overflow code snippets

open source projects, writing from scratch, modifying from the questions, and others, also follow the same proportions for both group.

The main difference we found are the responses regarding outdated code. The answerers in Group 1 have a more substantial percentage (61.2%) of being notified about the outdated code in their answers than Group 2 (47.1%). This may be caused by the amount of their Stack Overflow answers. Since Group-1 answerers have a higher reputation than Group-2 answerers, they possibly have given more answers on Stack Overflow. The higher amount of answers hence increase the chance of the code being outdated. Interestingly, although the percentage of outdated code notifications in Group 2 is lower, the percentage of the answerers who very frequently fixed their outdated code is higher than for Group 1.

Regarding the software license, the responses from the two groups agree with each other. The answerers in both groups show the same level of awareness to Stack Overflow CC BY-SA 3.0 license (62.1% and 62.3%). Similarly, the answerers in both groups neither include software license in their code snippets (98% and 99%) nor check of potential license conflicts between their code snippet's and Stack Overflow CC BY-SA 3.0 (69% and 69%).

The visitors' survey confirms the findings from the previous studies that Stack Overflow code snippets can be problematic [Acar et al., 2016, An et al., 2017]. Sixty-six percent of the visitors experienced a problem from reusing Stack Overflow code snippets ranging from incorrect solutions, outdated solutions, mismatched solutions, to buggy code. Although they are aware of the issues, half of them (56%) never report back to the Stack Overflow discussions. On the other hand, the visitors rarely give attributions to Stack Overflow when they reuse code snippets from the website, similar to the findings reported by Baltes et al. [2017]. The visitors are generally not aware of the CC BY-SA 3.0 license and more than half of them never check for potential license incompatibility when reusing Stack Overflow code snippets. We also found that 9% of the participants encountered legal issues by copying the code from Stack Overflow.

3.7 Threats to Validity

There are some potential threats to validity in this chapter. We separately discuss them in two aspects: internal and external validity.

3.7.1 Internal Validity

We only invited 607 developers to participate in the answerer survey. If all the developers are invited, which is almost impossible considering 7.6 million users on Stack Overflow, the results may be different. Nevertheless, we mitigate this threat to internal validity by selecting the participants based on their Stack Overflow reputation. This targeted participant selection ensures that we invite developers who have been actively involved in asking and answering questions on the site for an extended period of time, and refrain from inviting new users who merely answer a question or two. The highest reputation user invited to our survey answered 33,933 questions, and the lowest reputation user in our survey answered 116 questions.

The results from the answerer survey may not be completely accurate especially for RQ1 about the sources of code snippets in the Stack Overflow answers. It is socially not acceptable to copy code from a company in most cases. Even when a survey is anonymous, some answerers may hesitate to admit that.

We selected the participants for the visitor survey based on convenient sampling which could suffer from bias and outliers. We mitigate the threat by inviting different groups of participants ranging from the author's social media, technology news and media community, software engineering community (on Facebook), Java programmer discussion thread (`comp.lang.java.programmer`), and from the University of Molise.

3.7.2 External Validity

While the reputation is a good proxy to reflect the number and the quality of answers the developers have given, it might not cover all kinds of answerers and their experience on Stack Overflow. The answerers who have reputation lower than 6,999 were not invited to our study and their awareness and experience may differ from our findings. Hence, our conclusions may not be generalised to all Stack

Overflow answerers.

At least 39% of the participants in the visitor survey are from Thailand ([blognone.com](#) and some of the thesis author's contacts). Due to the working culture, answers from this group of developers may only represent software developers in Thailand. Similarly, our findings from the visitor survey may not be generalised to all Stack Overflow visitors. We alleviate this concern by inviting participants from other groups, e.g., the University of Molise in Italy, the `comp.lang.java.programmer` group, and the Software Engineering Facebook group. Nonetheless, due to the fewer number of the participants from the other groups compared to the thesis author's contacts and [blognone.com](#), the results may still suffer from biases.

3.8 Chapter Summary

This chapter presents the results from two online surveys of Stack Overflow answerers and visitors regarding their awareness and experience of outdated and potentially license-violating code snippets from answering and reusing code snippets on Stack Overflow.

The next chapter will apply code clone detection to empirically investigate the two issues of outdated code and software license incompatibility of code snippets on Stack Overflow. It will present code clones found between Stack Overflow and 111 open source projects, analyse the findings, and compare to the results from the surveys in this chapter.

Chapter 4

An Empirical Study of Online Code Clones and Their Toxicity

This chapter strengthens the results in the previous chapter by empirically studying online code clones, i.e., clones between software projects and Stack Overflow, and the two issues of outdated code and software license incompatibility. The chapter starts by discussing an experiment of online code clone detection between Stack Overflow and 111 open source projects. Then, it moves to present an analysis of the detected online code clones and their classifications. The chapter ends by describing potential problems from reusing outdated or license-violating Stack Overflow cloned code snippets.

4.1 Motivation

Similar to the previous chapter, a study in this chapter is motivated by the two issues of outdated code and license-violating code snippets on Stack Overflow. The process of posting and answering questions on Stack Overflow that involves the reuse (copying) of source code can be considered code cloning. Similar to traditional clones within software projects, software license violations from code cloning also happens within the context of online Q&A websites such as Stack Overflow. An et al. [2017] showed that 1,279 cloned snippets between Android apps and Stack Overflow potentially violate software licenses. Security is also among the main concerns when the code is copied from an online source. For example, Stack

Overflow helps developers to solve Android programming problems more quickly than other resources while, at the same time, offers less secure code than books or the official Android documentation [Acar et al., 2016].

4.1.1 Online Code Clones

We call code snippets that are copied from software systems to online Q&A websites (such as Stack Overflow) and vice versa as “**online code clones.**” There are two directions in creating online code clones: (1) code is cloned from a software project to a Q&A website as an example; or (2) code is cloned from a Q&A website to a software project to obtain a functionality, perform a particular task, or fixing a bug.

Similar to classic code clones, i.e., clones between software systems, online code clones can lead to license violations, bug propagation, an introduction of vulnerabilities, and re-use of outdated code. Unfortunately, online clones are difficult to locate and fix since the search space in online code corpora is larger and no longer confined to a local repository.

The previous chapter discusses a survey 201 high-reputation Stack Overflow answerers. The results of such a survey show that online code cloning occurs on Stack Overflow. Stack Overflow answerers frequently clone code from other locations, such as their personal projects, company projects, and open source projects, to Stack Overflow as a solution or as additional materials to a solution. The code cloning activity on Stack Overflow is obviously beneficial considered the popularity of Stack Overflow and its influence on software development [Ponzanelli et al., 2013, 2014, Park et al., 2014]. On the other hand, there is also a downside caused by low quality, defective, and harmful code snippets that are reused without awareness by millions of users [Zhang et al., 2018, Acar et al., 2016, Fischer et al., 2017].

As shown in the previous chapter (and copied below), a participant in our survey expresses his/her concerns about this:

The real issue is less about the amount the code snippets on SO than it is about the staggeringly high number of software “professionals” that

mindlessly use them without understanding what they’re copying, and the only slightly less high number of would-be professionals that post snippets with built-in security issues. A related topic is beginners who post (at times dangerously) misleading tutorials online on topics they actually know very little about. Think PHP/MySQL tutorials written 10+ years after mysql_ functions were obsolete, or the recent regex tutorial that got posted the other day on HackerNew (<https://news.ycombinator.com/item?id=14846506>). They’re also full of toxic code snippets.*

Although this activity of online code cloning is well-known, there are only a few empirical studies on the topic [An et al., 2017, Abdalkareem et al., 2017, Baltes et al., 2017], especially on finding the origins of the clones on Q&A websites. In this chapter, we tackle this challenge of establishing the existence of online code clones on Stack Overflow, investigate how they occur, and study the potential effects to software reusing them. Therefore, we mined Stack Overflow posts, detected online code clones, and analysed the clones to reveal “toxic code snippets.”

4.1.2 Toxic Code Snippets

Toxic code snippets mean code snippets that, after incorporating into software, the cost of paying back the technical debt exceeds the value it generates in the long run. Stack Overflow code snippets cloned from open source software or online sources can become toxic when they are (1) outdated, (2) violating their original software license, (3) exhibiting code smells, or (4) having security vulnerabilities.

In this chapter, we focus on the first two forms of toxic code snippets, outdated code and license-violating code, as these two problems are still underexplored compared to code smells [Tufano et al., 2015] and vulnerabilities [Acar et al., 2016, Fischer et al., 2017]. Moreover, as shown by the survey results in Chapter 3, Stack Overflow answerers and visitors expressed their concerns about these two problems. Outdated code snippets can be harmful since they are not up-to-date with their originals and may contain defects. License-violating code can be harmful because it leads to legal problems. Code snippets from open source projects usually fall

```

/* Code in Stack Overflow post ID 22315734 */
public int compare(byte[] b1, int s1, int l1, ...) {
    try {
        buffer.reset(b1, s1, l1); /* parse key1 */
        key1.readFields(buffer);
        buffer.reset(b2, s2, l2); /* parse key2 */
        key2.readFields(buffer);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return compare(key1, key2); /* compare them */
}

/* WritableComparator.java (2014-11-21) */
public int compare(byte[] b1, int s1, int l1, ...) {
    try {
        buffer.reset(b1, s1, l1); /* parse key1 */
        key1.readFields(buffer);
        buffer.reset(b2, s2, l2); /* parse key2 */
        key2.readFields(buffer);
        buffer.reset(null, 0, 0); /* clean up reference */
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return compare(key1, key2); /* compare them */
}

```

Figure 4.1: An example of the two code fragments of `WritableComparator.java`. The one from the Stack Overflow post 22315734 (left) is outdated when compared to its latest version in the Hadoop code base (right). Its Apache v.2.0 license is also missing.

under a specific software license, e.g., GNU General Public License (GPL). If they are cloned to Stack Overflow answers without the license, and then flow to other projects with conflicting licenses, legal issues may occur.

We would like to motivate the readers by giving two examples of toxic code snippets discovered by our study. The first example is an outdated and potentially license-violating online code clone fragment in an answer to a Stack Overflow question regarding how to implement `RawComparator` in Hadoop¹. Figure 4.1 shows—on the left—a code snippet embedded as a part of the accepted answer. The snippet shows how Hadoop implements the `compare` method in its `WritableComparator` class. The code snippet on the right shows another version of the same method, but at this time extracted from the latest version (as of 3rd October 2017) of

¹<http://stackoverflow.com/questions/22315734>

Hadoop. We can see that they both are highly similar except a line containing `buffer.reset(null, 0, 0);` which was added on 21st November 2014. The added line is intended for cleaning up the reference in the `buffer` variable and avoid excess heap usage (issue no. HADOOP-11323²). While this change has already been introduced into the `compare` method several years ago, the code example in Stack Overflow post is still unchanged. In addition, the original code snippet of `WritableComparator` class in Hadoop is distributed with Apache license version 2.0 while its cloned instance on Stack Overflow contains only the `compare` method and ignores its license statement on top of the file. There are two potential issues for this. First, the code snippet may appear to be under Stack Overflow's CC BY-SA 3.0 instead of its original Apache license. Second, if the code snippet is copied and incorporated into another software project with a conflicting license, a legal issue may arise.

The second motivating example of outdated online code clones with more disrupting changes than the first one can be found in an answer to a Stack Overflow question regarding how to format files sizes in a human-readable form. Figure 4.2 shows—on the left—a code snippet to perform the task from the `StringUtils` class in Hadoop. The code snippet on the right shows another version of the same method, but at this time extracted from the latest version of Hadoop. We can see that they are entirely different. The `humanReadableInt` method is rewritten on 5th February 2013 to solve an issue of a race condition (issue no. HADOOP-9252³).

The two toxic code snippets in our examples have been posted on 11th March 2014 and 9th April 2009 respectively. They have already been viewed 259 and 2,886 times⁴ at the time of writing this chapter (3rd October 2017). Our calculation finds that there will be a new viewer of the first toxic snippet approximately every 5 days compared to almost every day for the second one. Considering the popularity of Stack Overflow, which has more than 50 million developers visiting each month⁵,

²<https://issues.apache.org/jira/browse/HADOOP-11323>

³<https://issues.apache.org/jira/browse/HADOOP-9252>

⁴The number of views is for the whole Stack Overflow post but we use it as a proxy of the number of views the accepted answer receives because the question and the answer of the two motivation examples have a short gap of posting time (within the same day and four days after).

⁵Data as of 21st August 2017 from: <https://stackoverflow.com>

```

/* Code in Stack Overflow post ID 801987 */
public static String humanReadableInt(long number) {
    long absNumber = Math.abs(number);
    double result = number;
    String suffix = "";
    if (absNumber < 1024) {
    } else if (absNumber < 1024 * 1024) {
        result = number / 1024.0;
        suffix = "k";
    } else if (absNumber < 1024 * 1024 * 1024) {
        result = number / (1024.0 * 1024);
        suffix = "m";
    } else {
        result = number / (1024.0 * 1024 * 1024);
        suffix = "g";
    }
    return oneDecimal.format(result) + suffix;
}

/* StringUtils.java (2013-02-05) */
public static String humanReadableInt(long number) {
    return TraditionalBinaryPrefix.long2String(number, "", 1);
}

```

Figure 4.2: An example of the two code fragments of `StringUtils.java`. The one from the Stack Overflow post 801987 (left) is outdated when compared to its latest version in the Hadoop code base (right). The toxic code snippet is outdated code and has race conditions.

one toxic code snippet on Stack Overflow can spread and grow to hundreds or thousands of copies within only a year or two.

While research has mostly focused on reusing code snippets *from* Stack Overflow (e.g., Keivanloo et al. [2014], An et al. [2017], Yang et al. [2016]), fewer studies have been conducted on finding the origins of code examples copied *to* Stack Overflow. Finding the origins of code examples reveals the problem of toxic code snippets caused by outdated code and software license violations. It is equally important to studying the effects of reusing Stack Overflow code snippets because it gives insights into the root cause of the problem and lays a foundation of an automatic technique to detect and report toxic code snippets on Stack Overflow to developers in the future.

4.2 Contributions

This chapter makes the following primary contributions:

1. **A manual study of online code clones:** To empirically confirm the findings from the surveys, we used two clone detection tools to discover 2,289 similar code snippet pairs between 72,365 Java code snippets obtained from Stack Overflow’s accepted answers and 111 Java open source projects from the curated Qualitas corpus [Tempero et al., 2010]. We manually classified all of them.
2. **An investigation of toxic code snippets on Stack Overflow:** Our study shows that from the 2,289 online clones, at least 328 have been copied from open source projects or external online sources to Stack Overflow, and potentially violating software licenses. For 153 of them, we found evidence that they have been copied from a specific open source project. 100 of them were found to be outdated.
3. **An online code clone oracle:** The 2,289 manually investigated and validated online clone pairs are available for download⁶ and can be used as a clone oracle.

4.3 Empirical Study

We performed an empirical study of online code clones between Stack Overflow and 111 Java open source projects to answer the following research questions:

4.3.1 Research Questions

RQ1 (Online code clones): *To what extent is source code cloned between Stack Overflow and open source projects?* We quantitatively measured the number of online code clones between Stack Overflow and open source projects to understand the scale of the problem.

RQ2 (Patterns of online code clones): *How do online code clones occur?* We categorised online clones into seven categories allowing insights into how online code clones are created.

⁶<https://ucl-crest.github.io/cloverflow-web>

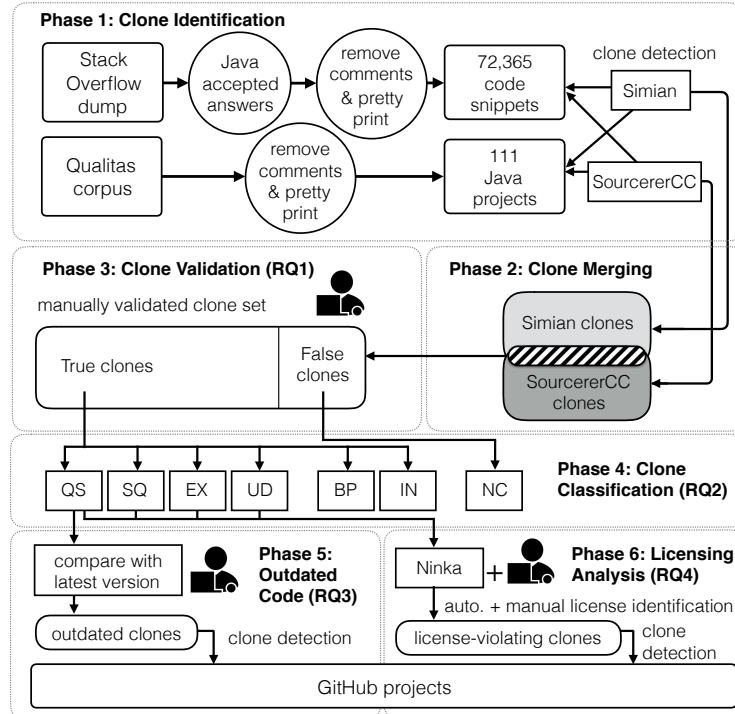


Figure 4.3: The Experimental framework

RQ3 (Outdated online code clones): *Are online code clones up-to-date compared to their counterparts in the original projects?* We were interested in the outdated Stack Overflow code examples since users are potentially reusing them.

RQ4 (Software license violation): *How often do license conflicts occur between Stack Overflow clones and their originals?* We investigated whether the reuse of online code clones can cause software developers to violate licenses.

To answer these four research questions, we perform an empirical study to study the online code clones between Stack Overflow and open source projects and their toxicity.

4.3.2 Online Code Clone Detection

We support the motivation and confirm the findings from the surveys in Chapter 3 by performing code clone detection between Stack Overflow answers and 111 Java

open source projects. We designed the study in 6 phases as depicted in Figure 4.3 where we build different data sets to answer RQ1 to RQ4.

4.3.2.1 Phase 1: Clone Identification

We rely on two source code data sets in this chapter: Java code snippets in answers on Stack Overflow and open source projects from the Qualitas corpus [Tempero et al., 2010], as detailed next.

Stack Overflow: We extracted Java code snippets from a snapshot of a Stack Overflow dump⁷ in January 2016. The data dump is in XML, and it contains information about posts (questions and answers). We were interested in code snippets embedded in posts which were located between `<code>...</code>` tags. A Stack Overflow thread contains a question and several answers. An answer can also be marked as an **accepted answer** by the questioner if the solution fixes his/her problem. We collected Java code snippets using two criteria. First, we only focused on code snippets in accepted answers. We chose the snippets in accepted answers because they actually solved the problems in the questions. Moreover, they are usually displayed just below the questions which makes them more likely to be reused than other answers. Second, we were only interested in code snippets of at least ten lines. Although the minimum clone size of six lines is usual in clone detection [Bellon et al., 2007, Wang et al., 2013b, Koschke et al., 2006], we empirically found that snippets of six lines contain a large number of boiler-plate code of getters/setters, `equal` or `hashCode` methods, which are not interesting for the study. Each snippet was extracted from the dump and saved to a file. Moreover, we filtered out irrelevant code snippets that were part of the accepted answers but were not written in Java by using regular expressions and manual checking. Finally, we obtained 72,365 Java code snippets containing 1,840,581 lines⁸ of Java source code. The median size of the snippets is 17 lines.

Open source systems: We selected the established **Qualitas** corpus [Tempero et al., 2010]. It is a curated Java corpus that has been used in several software

⁷<https://archive.org/details/stackexchange>

⁸Measured by cloc: <https://github.com/AlDanial/cloc>

Table 4.1: Stack Overflow and Qualitas datasets

Data set	No. of files	SLOC
Stack Overflow	72,365	1,840,581
Qualitas	166,709	19,614,083

engineering studies [Taube-Schock et al., 2011, Beckman et al., 2011, Vasilescu et al., 2011, Omar et al., 2012]. The projects in the corpus represent various domains of software systems ranging from programming languages to visualisation. We selected the release 20130901r of the Qualitas corpus containing 112 Java open source projects. This release contains projects with releases no later than 1st September 2013. We intentionally chose an old corpus from 2013 since we are interested in online code clones in the direction from open source projects to Stack Overflow. The 20130901r snapshot provides Java code that is more than 2 years older than the Stack Overflow snapshot, which is sufficiently long for a number of code snippets to be copied onto Stack Overflow and also to observe if clones become outdated. Out of 112 Qualitas projects, there is one project, *jre*, that does not contain Java source code due to its licensing limitation [Tempero et al., 2010] and is removed from the study. This resulted in 111 projects analysed in the study, for a total of 166,709 Java files containing 19,614,083 lines of code (see Table 4.1).

The median project size is 60,667 lines of code.

Clone Detection Tools: We use clone detection to discover online code clones. There are a number of restrictions in terms of choosing the clone detection tools for this chapter. The main restriction is due to the nature of code snippets posted on Stack Overflow, as most of them are incomplete Java classes or methods. Hence, a detector must be flexible enough to process code snippets that are not compilable or not complete blocks. Moreover, since the amount of code that has to be processed is in a scale of millions line of code (as shown in Table 6.1), a clone detector must be scalable enough to report clones in a reasonable amount of time. We have tried 7 state-of-the-art clone detectors including Simian [Harris, 2003], SourcererCC [Sajnani et al., 2016], NiCad [Roy and Cordy, 2008], CCFinder [Kamiya et al., 2002], iClones [Göde and Koschke,

2009], DECKARD [Jiang et al., 2007a], and PMD’s Copy/Paste Detector [CPD] against the Stack Overflow and Qualitas datasets. NiCad failed to parse 44,960 Stack Overflow snippets while PMD CPD failed to complete the execution due to lexical errors. iClones could complete its execution but skipped 367 snippets due to malformed blocks in Stack Overflow data sets. CCFinder reported 8 errors while processing the two data sets. Although Simian, SourcererCC, and DECKARD could successfully report clones, we decided to choose only Simian and SourcererCC due to their fast detection speed. Moreover, Simian and SourcererCC complement each other as SourcererCC’s clone fragments are always confined to method boundaries while Simian’s fragments are not.

Simian is a text-based clone detector that locates clones at line-level granularity and has been used extensively in several clone studies [Ragkhitwetsagul et al., 2016a, 2018a, Wang et al., 2013b, Mondal et al., 2011, Cheung et al., 2015, Krinke et al., 2010]. Furthermore, it offers normalisation of variable names and literals (strings and numbers) which enables Simian to detect literal clones (Type-1) and parameterised clones (Type-2) [Bellon et al., 2007].

SourcererCC is a token-based clone detector which detects clones at either function- or block-level granularity. It can detect clones of Type-1, -2 up to Type-3 (clones with added and removed statements) and offer scalability against large code corpus [Sajnani et al., 2016, Saini et al., 2016b, Yang et al., 2017].

We prepared the Java code in both datasets by removing comments and pretty-printing to increase the clone detection accuracy. Then, we deployed the two detectors to locate clones between the two datasets. For each Qualitas project, we ran the tools on the project’s code and the entire Stack Overflow data. Due to incomplete code blocks and functions typically found in Stack Overflow snippets, the built-in SourcererCC’s Java tokeniser could not parse 45,903 snippets, more than half of them. Nevertheless, the tool provides an option to plug in a customised tokeniser, so we developed a special Java tokeniser with assistance from the tool’s creators. The customised tokeniser successfully processed all Stack Overflow snippets.

Table 4.2: Configurations of Simian and SourcererCC

Tool	Configurations
Simian	Threshold=10, ignoreStringCase, ignoreCharacterCase, ignoreModifiers
SourcererCC	Functions, Minimum clone size=10, Similarity=80%

Simian did not provide an option to detect cross-project clones. Hence the Simian clone report was filtered to contain only clone pairs between Stack Overflow and Qualitas projects, removing all clone pairs within either Stack Overflow or Qualitas. SourcererCC can detect cross-project clones between two systems, so we did not filter the clones.

Clone Detection Configuration: We are aware of the effects of configurations to clone detection results and the importance of searching for optimised configurations in empirical clone studies [Svajlenko et al., 2014b, Wang et al., 2014, Ragkhitwetsagul et al., 2016b,a, 2018a]. However, considering the massive size of the two datasets and the search space of at least 15 Simian and 3 SourcererCC parameters, we are unable to search for the best configurations of the tools. Thus, we decided to configure Simian and SourcererCC based on their established default configurations chosen by the tools' creators as depicted in Table 4.2. The two clone detectors complemented each other by having Simian detecting literal copies of code snippets (Type-1) and SourcererCC detecting clones with renaming and added/deleted statements (Type-2, Type-3).

Nevertheless, we investigated a crucial parameter setting for clone detection: the minimum clone size threshold. Choosing a large threshold value can reduce the number of trivial clones (e.g., `equals`, `hashCode`, or getter and setter methods) and false clones in the analysis or the manual investigation phase [Sajnani et al., 2016], i.e., increasing precision. Nonetheless, it may create some false negatives. On the other hand, setting a low threshold results in a larger number of clone candidate pairs to look at, i.e., increasing recall, and a higher chance of getting false positives. Moreover, the large number of clone pairs hinder a full manual validation of the clones. Three threshold values, six, ten, and fifteen lines, were

chosen for our investigation. We started the investigation by using a threshold value of six lines, a well-accepted minimum clone size in clone benchmark [Bellon et al., 2007]. Simian reported 67,172 clone candidate pairs and SourcererCC reported 7,752 clone candidate pairs. We randomly sampled 382 pairs from the two sets for a manual check. This sample number was a statistically significant sample with a 95% confidence level and $\pm 5\%$ confidence interval. The thesis author investigated the sampled clone pairs and classified them into three groups: non-clones, trivial clones (`equals`, `hashCode`, or getter and setter methods), and non-trivial clones. The manual check found 26 non-clone pairs, 322 trivial clone pairs, and 34 non-trivial clone pairs. Next, we increased the threshold to ten lines, another well-established minimum clone size for large-scale data sets [Sajnani et al., 2016], and retrieved 721 clone pairs from Simian and 1,678 clone pairs from SourcererCC. We randomly sampled and manually checked the same amount of 382 pairs and found 27 non-clone pairs, 253 trivial clone pairs, and 102 non-trivial clone pairs. Then, we increased the threshold further to fifteen lines and retrieved 196 clone pairs from Simian and 1,230 clone pairs from SourcererCC. The manual check of the 382 randomly sampled pairs revealed zero non-clone pairs, 298 trivial clone pairs, and 83 non-trivial clone pairs.

The findings from the three threshold values show that selecting the minimum clone size of ten lines was preferred over six and fifteen lines. First, it generated a fewer number of clone pairs than using six lines, which made the manual clone investigation feasible. Second, it preserved the highest number of non-trivial clone pairs.

The number of online clone pairs reported using the minimum clone size of 10 lines are presented in Table 4.3. Simian reports 721 clone pairs while SourcererCC reports 1,678 clone pairs. The average clone size reported by Simian is 16.61 lines which is slightly smaller than SourcererCC (17.86 lines).

4.3.2.2 Phase 2: Clone Merging

Clones from the two detectors can be duplicated. To avoid double-counting of the same clone pair, we adopted the idea of **clone agreement** which has been used in

Table 4.3: Number of online clones reported by Simian and SourcererCC

Tool	Total clone pairs	Average clone size
Simian	721	16.61
SourcererCC	1,678	17.86

clone research studies [Funaro et al., 2010, Wang et al., 2013b, Ragkhitwetsagul et al., 2016b] to merge clones from two data sets. Clone pairs agreed by both clone detection tools have a high likelihood to be duplicate and must be merged. To find agreement between two clone pairs reported by two different tools, we used the clone pair matching metric proposed by Bellon et al. [2007]. Two clone pairs that have a large enough number of overlapping lines can be categorised as either a good-match or an ok-match pair with a confidence value between 0 and 1. Although good-match has a stronger agreement than ok-match, we choose the ok-match criterion as our clone merging method because it depends on clone containment and does not take clone size into account. Clone containment suits our online code clones from two tools, Simian (line-level) and SourcererCC (method-level), better because Simian’s clone fragments can be smaller or bigger than a method while SourcererCC’s clone fragments are always confined to a method boundary.

We follow Bellon’s original definitions of ok-match [Bellon et al., 2007], which are based on how much two clone fragments CF are contained in each other:

$$\text{contained}(CF_1, CF_2) = \frac{|\text{lines}(CF_1) \cap \text{lines}(CF_2)|}{|\text{lines}(CF_1)|}$$

A clone pair CP is formed by two clone fragments CF_1 and CF_2 , i.e., $CP = (CF_1, CF_2)$ and the *ok-value* of two clone pairs is defined as

$$\begin{aligned} \text{ok}(CP_1, CP_2) &= \min(\max(\text{contained}(CP_1.CF_1, CP_2.CF_1), \\ &\quad \text{contained}(CP_2.CF_1, CP_1.CF_1)), \\ &\quad \max(\text{contained}(CP_1.CF_2, CP_2.CF_2), \\ &\quad \text{contained}(CP_2.CF_2, CP_1.CF_2))) \end{aligned}$$

Two clone pairs CP_1 and CP_2 are called an *ok-match*(t) iff, for threshold $t \in [0, 1]$ holds

$$ok(CP_1, CP_2) \geq t$$

The threshold t is crucial for the ok-match because it affects the number of merged clone pairs. Setting a high t value will result in a few ok-match clone pairs and duplicates of the same clone pairs (which are supposed to be merged) may appear in the merged clone set. On the other hand, setting a low t value will result in many ok-match clone pairs, and some non-duplicate clone pairs may be accidentally merged by only a few matching lines. In order to get an optimal t value, we did an analysis by choosing five t values of 0.1, 0.3, 0.5, 0.7, 0.9 and studied the merged clone candidates. By setting $t = 0.7$ according to Bellon's study, we found 97 ok-match pairs reported. On the other hand, setting t to 0.1, 0.3, 0.5, and 0.9 resulted in 111, 110, 110, and 94 ok-matched pairs respectively. Since the clone pairs of $t = 0.1$ were the superset of other sets, we manually checked all the 111 reported pairs. We found one false positive pair and 110 true positive pairs. By raising the t to 0.3 and 0.5, we got rid of the false positive pair and still retained all the 110 true positive pairs. All the clone pairs of $t = 0.7$ (97) and $t = 0.9$ (94) were also true positives due to being a subset of $t = 0.5$. However, since there were fewer merged clone pairs, we ended up leaving some duplicates of the same clones in the final merged clone set. With this analysis, we can see that setting the threshold t to 0.1 is too relaxed and results in having false positive ok-match pairs, while setting the t to 0.7 or 0.9 is too strict. Thus, we decided to select the t value at 0.5.

Using the ok-match criterion with the threshold t of 0.5 similar to Bellon's study [Bellon et al., 2007], we merge 721 clone pairs from Simian and 1,678 clone pairs from SourcererCC into a single set of 2,289 online clone pairs. There are 110 common clone pairs between the two clone sets as depicted in Figure 4.4. The low number of common clone pairs is due to SourcererCC reporting clones with method boundaries while Simian is purely line-based.

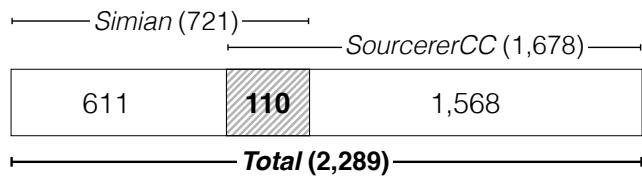


Figure 4.4: The result from clone merging using Bellon's ok-match criterion

4.3.2.3 Phase 3–4: Validation and Classification

We used the 2,289 merged clone pairs for manual validation and online clone classification. The validation and classification of the pairs were done at the same time. The clone validation process (phase 3 in Figure 4.3) involves checking if a clone pair is a true positive or a false positive. Moreover, we are also interested in the patterns of code cloning so we can gain more insights into how these clones are created (phase 4 in Figure 4.3).

Manual investigation: To mitigate the human error, we deployed two people in the manual clone investigation process. The author of the thesis and a software engineering research student in the Centre for Research on Evolution, Search and Testing (CREST), who is familiar with code clones, took the role of the investigators performing a manual validation and classification of the merged clone pairs. The two investigators separately went through each clone pair candidate, looked at the clones, and decided if they are a true positive or a false positive and classified them into an appropriate pattern. After the validation, the results from the two investigators were compared. There were 338 (15%) conflicts between true and false clones (QS, SQ, EX, UD, BP, IN vs. NC). The investigators looked at each conflicting pair together and discussed until a consensus was made. Another 270 pairs (12%) were conflicts in the classification of the true clone pairs. Among these pairs, 145 conflicts were caused by one investigator being more careful than the other and being able to find evidence of copying while the other could not. Thus, resolving the conflicts lead to a better classification, i.e., from UD to QS or EX.

The online cloning classification patterns: We studied the eight patterns of cloning from Kapser and Godfrey [2006, 2008] and performed a preliminary study to evaluate its applicability to our study. We tried to classify 697 online

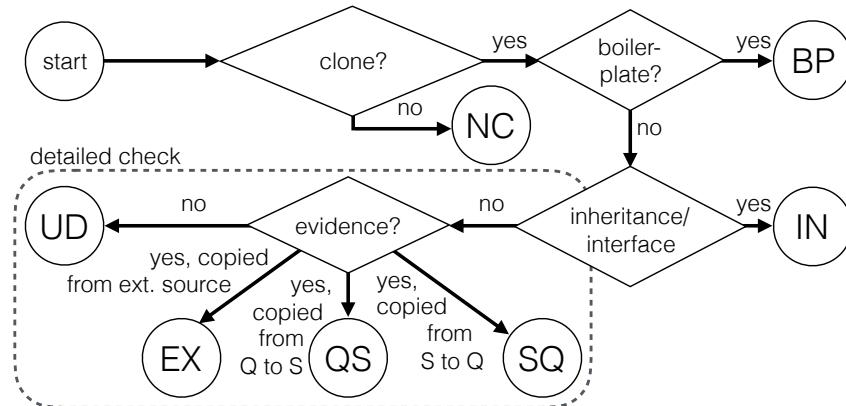
Table 4.4: The seven patterns of online code cloning

Patt.	Description
QS	Cloned from Qualitas project to Stack Overflow ($Q \rightarrow S$)
SQ	Cloned from Stack Overflow to Qualitas project ($S \rightarrow Q$)
EX	Cloned from an external source to Stack Overflow ($X \rightarrow S$)
UD	Cloned from each other or from an external source outside the project (unknown)
BP	Boiler-plate or IDE auto-generated
IN	Inheritance, interface implementation
NC	Not clones

clone pairs from the reported clones in phase 1 using Kapser’s cloning patterns. We found that Kapser’s patterns are too broad for our study and a more suitable and fine-grained classification scheme is needed. After a preliminary study, we adopted one of Kapser’s cloning patterns, *boiler-plate code*, and defined six new cloning patterns. The seven patterns include QS, SQ, EX, UD, BP, IN, and NC as presented in Table 4.4. Pattern QS (**Qualitas to Stack Overflow**) represents clones that have evidence of being copied from a Qualitas project to Stack Overflow. The evidence of copying can be found in comments in the Qualitas source code or in the Stack Overflow post’s contents. Pattern SQ (**Stack Overflow to Qualitas**) is cloning, with evidence, in the opposite direction from Stack Overflow to a Qualitas project. Pattern EX (**External Sources**) is cloning that has evidence of copying from a single or multiple external sources to Stack Overflow, and possibly also to a Qualitas project. Pattern UD (**Unknown Direction**) is cloning that creates identical or highly similar clones between Qualitas and Stack Overflow but where we could not find any attribution of copying. Pattern BP (**Boiler-Plate**) represents clones containing boiler-plate. We define three cases of boiler-plate code and use in our classification as shown in Table 4.5. Our definition is specific to Java and more suitable to our study than the general definition in Kapser’s [Kapser and Godfrey, 2008]. Pattern IN (**Inheritance/Interface**) is cloning by inheritance of the same super class or implementation of the same interface. These two activities usually result in similar overriding methods. The last pattern, NC (**Not Clones**), represents false

Table 4.5: The definition of boiler-plate code

Type	Description
API constraints	Similar code fragments are created because of a constraint by an API. For example, reading and writing to database using JDBC, reading and writing a file in Java.
Templating	An optimised or stable code fragment is reused multiple times. This also includes auto-generated code by IDE.
Design patterns	Java design patterns suggest a way of implementing similar pieces of code. For example, getters, setters, equals, hashCode, and toString method.

**Figure 4.5:** Online code clone classification process

clone pairs. These are mainly false positive clones from the clone detectors such as similar try-catch statements.

The classification of the filtered online clone pairs followed the steps depicted in Figure 4.5. First, we looked at a pair of clone fragments to see their similarity. If they were accidentally similar clones after code normalisation or false positive clones from the clone detection tools, we classified the pair into **NC**. If the two fragments were boiler-plate code, the pair was classified into **BP**. If they implemented the same interface or inherited the same class and shared similar overriding methods, the pair was classified into **IN**. If the pair was not **BP**, **IN**, or **NC**, we started a detailed investigation. We checked the corresponding Stack Overflow post, read through it carefully and looked for any evidence mentioning code copying. If evidence of copying had been found from a Qualitas project, the pair was classified in **QS**. In several occasions, we used extra information

such as the questions' contents, the name of posters, and the tags to gain a better understanding. On the other hand, if the source code from the Qualitas project mentioned copying from Stack Overflow, the pair was classified into **SQ**. If there was evidence of copying from an external source instead of a Qualitas project, the pair was classified as **EX**. Lastly, if there was no evidence of copying in any direction but the clone fragments were highly similar, we classified them into **UD**.

4.3.2.4 Phase 5: Outdated Clones

Outdated code occurs when a piece of code has been copied from its origin to another location, and later the original has been updated [Xia et al., 2014]. Usually, code clone detection is used to locate clone instances and update them to match with the originals [Bellon et al., 2007]. However, online code clones are more difficult to detect than in regular software projects due to its large search space and a mix of natural and programming languages combined in the same post.

To search for outdated online code clones, we focused on the **QS** clone pairs that were cloned from Qualitas to Stack Overflow and compared them with their latest versions. We downloaded the latest version of the Qualitas projects from their repositories on 3 October 2017. For each **QS** online clone pair, we used the clone from Qualitas as a proxy. We searched for its latest version by the file name and located the cloned region in the file based on the method name or specific code statements. We then compared the Stack Overflow snippet to its latest version line-by-line to find if any change has been made to the source code. We also made sure that the changes did not come from the modifications made to the Stack Overflow snippets by the answerers but from the updates in the projects themselves. When we found inconsistent lines between the two versions, we used `git blame` to see who modified those lines of code and the timestamps. We also read commit messages and investigated the issue tracking information if the code change is linked to an automatic issue tracking system, such as Jira or BugZilla to gain insights into the intent behind the change.

Lastly, we searched for the outdated code snippets in 130,719 GitHub projects to see how widespread is the outdated code in the wild. We mined GitHub based

on the number of stars the projects received, which indicated their popularity. We relied on GitHub API to query the project metadata before cloning them. Since GitHub API returned only top 1,000 projects at a time for each query, we formulated the query to retrieve most starred projects based on their sizes. The project size range started from 1KB to 2MB with 1KB step size, and the last query is for all the remaining projects that were larger than 2MB. With this method, we retrieved the top 1,000 most starred projects for each project size. As a result, we cloned 130,719 GitHub projects ranging from 29,465 stars to 1 star. A clone detection was then performed between the outdated code snippets and the GitHub projects. We selected SourcererCC with the same settings (see Table 4.2) for this task because it could scale to a large-scale data set, while Simian could not. Finally, we analysed the clone reports and manually checked the clones.

4.3.2.5 Phase 6: Licensing Analysis

Software license plays an important role in software development. Violation of software licenses impacts software delivery and also leads to legal issues [Sprigman, 2015]. One can run into a licensing issue if he or she integrates third-party source code into their software without checking. A study by An et al. [2017] reports 1,279 cases of potential license violations between 399 Android apps and Stack Overflow code snippets.

We analysed licensing conflicts of the online clones in the **QS**, **EX**, and **UD** set. The licenses were extracted by *Ninka*, an automatic license identification tool [German et al., 2010]. Since Ninka works at a file level, we reported the findings based on Stack Overflow snippets and Qualitas source files instead of the clone pairs (duplicates were ignored). For the ones that could not be automatically identified by Ninka and had been reported as `SeeFile` or `Unknown`, we looked at them manually to see if any license can be found. For EX clone pairs that are cloned from external sources such as JDK or websites, we manually searched for the license of the original code. Lastly, we searched for occurrences of the license-conflicting online clones in GitHub projects using the same method as in the outdated clones.

Table 4.6: Investigated online clone pairs and corresponding snippets and Qualitas projects

Set	Pairs	Snippets	Projects	Cloned ratio
Reported clones	2,289	460	59	53.28%
TP from manual validation	2,063	443	59	54.09%

4.4 Results and Discussion

We follow the 6 phases in the experimental framework (Figure 4.3) to answer the four research questions. To answer RQ1, we rely on the number of manually validated true positive online clone pairs in phase 3. We use the results of the manual classification by the seven patterns of online code cloning to answer RQ2 (phase 4). For RQ3, we looked at the true positive clone pairs that are classified as clones from Qualitas to Stack Overflow and checked if they have been changed after cloning (phase 5). Similarly, for RQ4, we looked at the license of each clone in the pattern QS, EX, UD and checked for a possibility of license violation (phase 6).

4.4.1 RQ1: Online Code Clones

To what extent is source code cloned between Stack Overflow and open source projects?

The statistics on clones obtained from the merged clone data set are presented in Table 4.6. Simian and SourcererCC reported clones in 460 snippets, approximately 0.6% of the 72,365 Stack Overflow snippets, associated with 59 Qualitas projects. For the cloned Stack Overflow snippets, the average ratio of cloned code is 53.28% (i.e., the number of cloned lines of the cloned Stack Overflow snippet).

Lastly, during the manual investigation of 2,289 clone pairs, we identified 226 pairs as being accidental clones (NC), i.e., false positives. After removing them, the set still contains 2,063 true positive clone pairs between 443 Stack Overflow snippets and 59 Qualitas projects. The average cloned ratio for the true positive clone pairs is 54.09%.

To answer RQ1, we found 2,063 manually confirmed clone pairs between 443

Table 4.7: Classifications of online clone pairs.

Set	QS	SQ	EX	UD	BP	IN	NC	Total
Before consolidation	247	1	197	107	1,495	16	226	2,289
After consolidation	153	1	109	65	216	9	53	606

Stack Overflow code snippets and 59 Qualitas projects.

4.4.2 RQ2: Patterns of Online Code Cloning

How do online code clones occur?

We performed a manual classification of the 2,289 merged clone pairs by following the classification process in Figure 4.5. The classification results are shown in Table 4.7 and explained in the following.

QS: Qualitas → Stack Overflow. We found 247 online clone pairs with evidence of cloning from Qualitas projects to Stack Overflow. However, we observed that, in some cases, a cloned code snippet on Stack Overflow matched with more than one code snippet in Qualitas projects because of code cloning inside Qualitas projects themselves. To avoid double counting of such online clones, we consolidated multiple clone pairs having the same Stack Overflow snippet, starting line, and ending line into a single clone pair. We finally obtained 153 QS pairs (Table 4.7) having unique Stack Overflow code snippets and associated with 23 Qualitas projects listed in Table 4.8. The most cloned project is `hibernate` with 23 clone pairs, followed by `eclipse` (21 pairs), `jung2` (19 pairs), `spring` (17 pairs), and `jfreechart` (13 pairs). The clones are used as examples and are very similar to their original Qualitas code with limited modifications. Most of them have a statement in the Stack Overflow post saying that the code is “copied,” “borrowed” or “modified” from a specific file or class in a Qualitas project. For example, according to the motivating example in Figure 4.1, we found evidence in the Stack Overflow Post 22315734 saying that “*Actually, you can learn how to compare in Hadoop from WritableComparator. Here is an example that borrows some ideas from it.*”

We analysed the time it took for the clones to appear from Qualitas projects to Stack Overflow. The clone ages were calculated by counting the number of months

Table 4.8: Qualitas projects associated with QS and UD online clone pairs

QS		UD	
Project	Pairs	Project	Pairs
hibernate	23	netbeans	11
eclipse	21	eclipse	8
jung2	19	jstock	5
spring	17	compiere	5
jfreechart	13	ireport	4
hadoop	10	jmeter	4
tomcat	8	jung2	3
log4j	8	jhotdraw	3
struts2	5	c-jdbc	3
weka	4	log4j	3
lucene	4	wct	2
poi	3	hibernate	2
junit	3	tomcat	2
jstock	2	spring	1
jgraph	2	rssowl	1
jboss	2	mvnforum	1
jasperreports	2	jfreechart	1
compiere	2	jboss	1
jgrapht	1	hadoop	1
itext	1	geotools	1
c-jdbc	1	freemind	1
ant	1	findbugs	1
antlr4	1	cayenne	1

between the date of each Qualitas project and the date the answer was posted on Stack Overflow as shown in Figure 4.6. We found that, on average, it took the clones around two years since they appeared in Qualitas projects to appear on Stack Overflow answers. Some of the clones appeared on Stack Overflow almost at the same time as the original, while the oldest clones took around five years.

SQ: Stack Overflow → Qualitas. We found one pair with evidence of cloning from Stack Overflow post ID 698283 to `POIUtils.java` in `jstock` project. The user who asked the question on Stack Overflow is an author of `jstock`. The question is about determining the right method to call among seven overloading methods of `setCellValue` during runtime. We could not find evidence of copying or attribution to Stack Overflow in `jstock`. However, considering that the 25 lines of code of

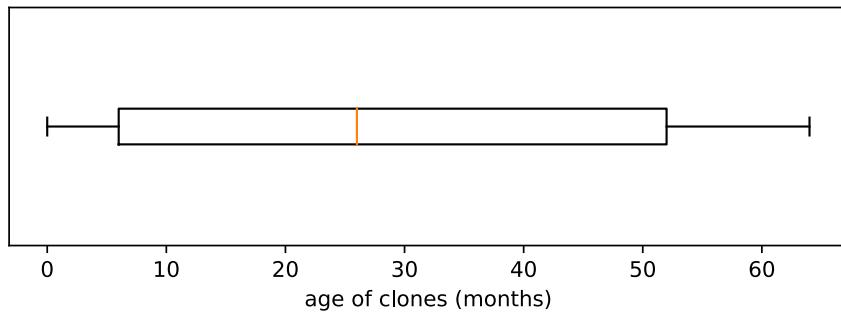


Figure 4.6: Age of QS online code clones.

```

private Method findMethodToInvoke(Object test) {
    Method method = parameterTypeMap.get(test.getClass());
    if (method != null) {
        return method;
    }

    // Look for superclasses
    Class<?> x = test.getClass().getSuperclass();
    while (x != null && x != Object.class) {
        method = parameterTypeMap.get(x);
        if (method != null) {
            return method;
        }
        x = x.getSuperclass();
    }

    // Look for interfaces
    for (Class<?> i : test.getClass().getInterfaces()) {
        method = parameterTypeMap.get(i);
        if (method != null) {
            return method;
        }
    }
    return null;
}

```

Figure 4.7: The `findMethodToInvoke` that is found to be copied from Stack Overflow post 698283 to POIUtils class in jstock.

`findMethodToInvoke` method depicted in Figure 4.7 in Stack Overflow is very similar to the code in `jstock` including comments, it is almost certain that the two code snippets form a clone pair. In addition, the Stack Overflow answer was posted on 30 March 2009, while the code in `POIUtils` class in `jstock` was committed to GitHub on the next day of 31 March 2009.

This very low number of SQ clone pair is very likely due to the age of the

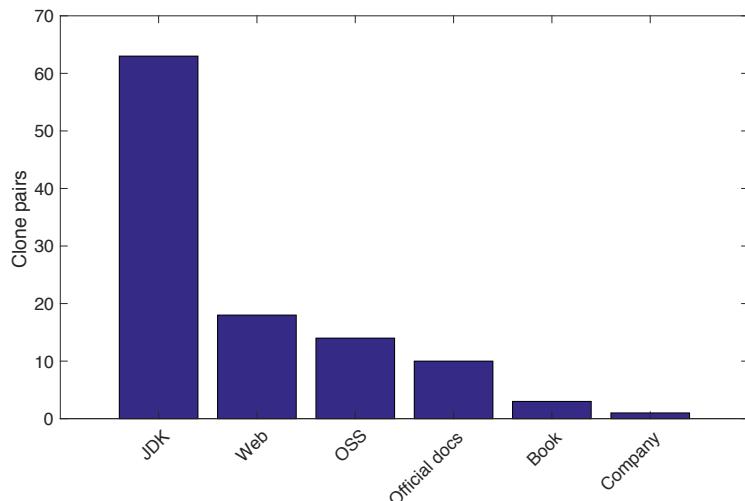


Figure 4.8: Original sources of EX clone pairs

Qualitas corpus as another study [An et al., 2017] showed the presence of clones from Stack Overflow in newer open source data sets. This is expected and comes from our experimental design since we are more interested in cloning from Qualitas to Stack Overflow.

EX: External Sources. We found 197 clone pairs from external sources to Stack Overflow. After consolidating duplicated SO snippets due to multiple intra-clone instances in Qualitas, we obtained 109 EX pairs. We found evidence of copying from an external source to both Stack Overflow and Qualitas in 49 pairs. Each of the pairs contains statement(s) pointing to the original external location of the cloned code in Qualitas and Stack Overflow. Besides, we found evidence of copying from an external source to Stack Overflow, but not in Qualitas, in 60 pairs. Our analysis shows that the external sources fall into six groups as displayed in Figure 4.8. There are 63 EX online clone pairs copied from source code of Java SDK (e.g., `java.util`, `javax.swing`, `javax.servlet`), 18 pairs from websites, 14 pairs from open source systems not in Qualitas (e.g., Mozilla Rhino), 10 pairs from Java official documentation by Sun Microsystems or Oracle, 3 pairs from books, and 1 pair from a company project. For example, Stack Overflow Post 9549009 contains a code comment saying “*Copied shamelessly from org.bouncycastle.crypto.generators.PKCS5S2ParametersGenerator*” which is an open source project not included in the Qualitas corpus. Post 92962 includes

a `VerticalLabelUI` class with a license statement showing that it is developed by a private company called Sapient. Post 12879764 has a text saying “*Code modified and cleaned from the original at Filthy Rich Clients.*” which is a book for developing animated and graphical effects for desktop Java applications. Another example is a copy of the code from a website in post 15260207. The text surrounding source code reads “*I basically stole this from the web and modified it slightly... You can see the original post here (<http://www.java2s.com/Code/Java/Swing-JFC/DragListDemo.htm>).*”. Interestingly, the code is actually a copy from Sun Microsystems.

These findings complement a study of clones between software projects [Svaljlenko et al., 2014b]. We found that cloning can also happen among different sources on the Internet just like software projects. There are 18 clone pairs that originated from programming websites including `www.java2s.com` and `exampledepot.com`. Moreover, there is one snippet which comes from a research website. We found that a snippet to generate graphical *Perlin noise* is copied from NYU Media Research Lab (`http://mrl.nyu.edu/~perlin/noise/`) website and is used in both Stack Overflow answer and the aoi project with attribution.

UD: Unknown Direction. We found 107 online clone pairs, reduced to 65 pairs after consolidating the clones, with no evidence of cloning between Qualitas and Stack Overflow but with a high code similarity that suggests cloning. The most cloned project is `netbeans` with 11 clone pairs. Most of the clones are a large chunk of code handling GUI components. Although these GUI clones might be auto-generated by IDEs, we did not find any evidence. The second most cloned project is `eclipse` (8 pairs), followed by `jstock` (5 pairs), a free stock market software, and `compiere`, a customer relationship management (CRM) system.

BP: Boiler-Plate. There were a large amount of boiler-plate clone pairs found in this study. We observed 1,495 such clone pairs and 216 after consolidation. The BP clone pairs account for 65% of all clone pairs we classified. The majority of them are `equals()` methods.

IN: Inheritance/interface. There were 16 clone pairs, 9 pairs after consol-

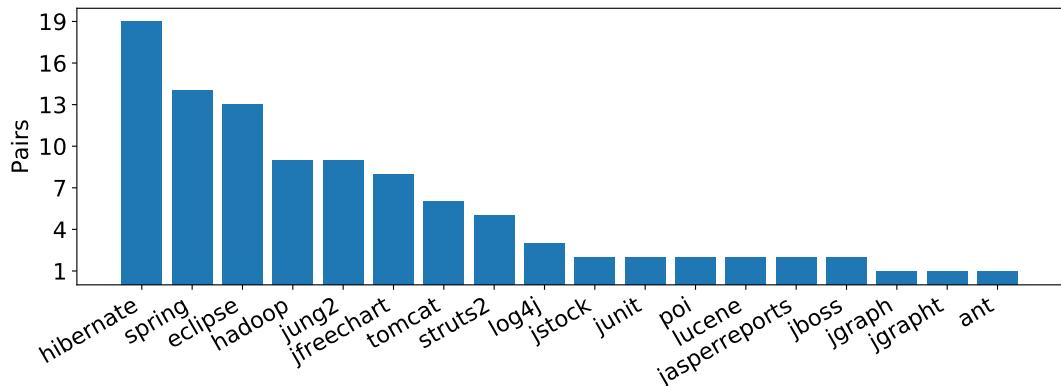


Figure 4.9: Outdated QS online clone pairs group by projects

idation, found to be similar because they implement the same interface or inherit from the same class. An example is the two implementations of a custom data type which implements `UserType`. They share similar `@Override` methods of `deepCopy`, `isMutable`, `assemble`, `disassemble`, and `replace`.

NC: Not Clones. There were 226 non-clone pairs, 53 after consolidation. Mainly, they are false positive clones caused by code normalisation and false Type-3 clones from SourcererCC. Examples of the NC clone instances include `finally` or `try-catch` clauses that were accidentally the same due to their tiny sizes, and similar `switch-case` statements.

To answer RQ2, we found 153 pairs with strong evidences to be cloned from 23 Qualitas projects to Stack Overflow, 1 pair was cloned from Stack Overflow to Qualitas, and 109 pairs were found to be cloned to Stack Overflow from external sources. However, the largest amount of the clone pairs between Stack Overflow and Qualitas projects are boiler-plate code (216), followed by 65 clone pairs with no evidence that the code has actually been copied, and 9 pairs of clones due to implementing the same interface or inheriting the same class.

4.4.3 RQ3: Outdated Online Code Clones

Are online code clones up-to-date compared to their counterparts in the original projects?

We discovered 100 outdated online clone pairs out of 153 pairs. As shown in Figure 4.9, `hibernate` has the highest number of 19 outdated pairs, followed by

Table 4.9: Six code modification types found when comparing the outdated clone pairs to their latest versions

Modification	Occurrences
Statement modification	50
Statement addition	28
Statement removal	18
Method signature change	16
Method rewriting	15
File deletion	14

14 from `spring`, 13 from `eclipse`, and 9 from `hadoop`. Besides the two examples of outdated code in `WritableComparator` and `StringUtils` class from `hadoop` shown in Figure 4.1 and Figure 4.2, we also found a few outdated code elements which contained a large number of modifications. For example, the code snippet in Stack Overflow post 23520731 is a copy of `SchemaUpdate.java` in `hibernate`. The code has been heavily modified on 5 February 2016.

We analysed code modifications which made Stack Overflow code outdated by going through commits and git blame information. The six code modification types found in the 100 outdated online clone pairs are summarised in Table 4.9. They include statement addition, statement modification, statement removal, method rewriting, API change (changing in method signature), and file deletion. We occasionally found multiple code modifications applied to one clone pair at the same time but at a different location. The most often code change found is statement modification (50 occurrences), followed by statement addition (28 occurrences), statement removal (18 occurrences), change of method signature, i.e., API change (16 occurrences), and method rewriting (15 occurrences). Moreover, in the 101 outdated pairs, we found 15 “dead” snippets. These snippets cannot be located in the latest version of the projects. For example, the snippet in Stack Overflow post 3758110, a copy of `DefaultAnnotationHandlerMapping.java` in `spring`, was deleted in the commit `02a4473c62d8240837bec297f0a1f3cb67ef8a7b` on 20 January 2012, two years after it was posted.

Moreover, using the information in git commit messages, we can associate

each change to its respective issues in an issue tracking system, such as Bugzilla or Jira. We found that in 58 cases, the cloned code snippets on Stack Overflow were changed because of a request in the issue tracking system. Since issue tracking systems are also used, besides bug reports, for feature request and feature enhancements, having an issue tracking ID can reflect that most of the changes are important and not only a superficial fix such as code formatting. The intents behind the changes are grouped into six categories as shown in Section 4.4.3. Enhancement is the majority intent accounting for 65 of the 100 outdated code (65%), followed by code depreciation (15%). There were 10 outdated code snippets (10%) caused by bug fixing. The rest of the changes are because of code refactoring (6%), changing coding style (3%), and the data format change (2%). Not all outdated code are toxic. However, the 10 buggy and outdated code snippets we found are toxic and are harmful to reuse.

Table 4.10 shows examples of the outdated online clones on Stack Overflow. The table displays information about the clones from both Stack Overflow and Qualitas side including the dates. We summarise the changes that make the clones outdated into three types, modified/added/deleted statements (S), file deletion (D), and method rewriting (R), along with the issue tracking number and the date of the change. The complete set of 100 outdated online clones can be found in Table B.1 and Table B.2 in Appendix B.

We performed a detailed investigation of the 100 outdated answers on Stack Overflow, on 6 May 2018, approximately two years after the snapshot we analysed was created to look for any changes, warnings, or mitigations made to the outdated code snippets. We investigated the answers on three aspects: newer answers, higher-voted answers, and comments on the outdated answers. We found 34 posts which contained newer answers and 5 posts which contained answers with a higher number of votes than the outdated answers. However, 99 of the 100 outdated answers were still marked as accepted answers.

Table 4.10: Examples of the outdated QS online clones

Stack Overflow				Qualitas				Changes			
Post	Date	Project	Ver.	File		Start	End	Date	Issue ID	Type*	Date
2513183	25-03-10	eclipse	4.3	GenerateToStringAction.java	113	166	05-06-13	Bug 439874	S	17-03-15	
22315734	11-03-14	hadoop	1.0.0	WritableComparator.java	44	54	25-08-11	HADOOP-11323	S	20-11-14	
23520731	07-04-14	hibernate	4.2.2	SchemaUpdate.java	115	168	22-05-13	HHH-10458	S	05-02-16	
18232672	14-08-13	log4j	1.2.16	SMTPAppender.java	207	228	31-03-10	Bug 44644	R	18-10-08	
17697173	17-07-13	lucene	4.3.0	SlowSynonymFilterFactory.java	38	52	06-04-13	LUCENE-4095	D	31-04-12	
21734562	12-02-14	tomcat	7.0.2	FormAuthenticator.java	51	61	04-08-10	BZ 59823	R	04-08-16	
12593810	26-09-12	poi	3.6	WorkbookFactory.java	49	60	07-12-09	57593	R	30-04-15	
8037824	07-11-11	jasperreports	3.7.4	JRVerifier.java	1221	1240	31-05-10	N/A	D	20-05-11	
3758110	21-09-10	spring	3.0.5	DefaultAnnotation	78	92	20-10-10	SPR-14129	D	20-01-12	
14019840	24-12-12	struts	2.2.1	DefaultActionMapper.java	273	288	17-07-10	WW-4225	S	18-10-13	

* S: modified/added/deleted statements, D: file has been deleted, R: method has been rewritten completely

Intent	Detail	Amount
Enhancement	Add or update existing features	64
Deprecation	Delete dead/deprecated code	15
Bug	Fix bugs	10
Refactoring	Refactor code for better design	6
Coding style	Update to a new style guideline	3
Data change	Changes in the data format	2

Table 4.11: Intents of code changes in 100 outdated code snippets

Clones	Amount
<i>Found in Qualitas GitHub repos</i>	13
<i>Found in other project repos</i>	
Exact copy (outdated)	47
Non-exact copy	32
Total	102

Table 4.12: Clones of the 100 Stack Overflow outdated code snippets in 130,719 GitHub projects

For the comments, we check if there is any comment to mitigate or point out the toxicity of the outdated code snippets. We found that, out of 100 answers, 6 answers had a comment saying the code in the answer is outdated or containing issues, such as “*spring 3.1 stuff*”, “*...tried this but having connect exception – javax.mail.MessagingException: Could not connect to SMTP host: smtp.gmail.com, port: 465*”, “*You should add a buffer.reset(null, 0, 0); at the end of the try block to avoid excess heap usage (issue no. HADOOP-11323)*” or “*.. I do not have experience with new versions of hibernate for a long time. But previously without clean you could receive some unexpected results. So I suggest to try different approaches or even check latest documentation*”. The 6 outdated code snippets were still not fixed, but the comments themselves may help to warn some of the Stack Overflow users.

Then, we performed code clone detection between the 100 outdated code snippets and 130,719 GitHub projects. We found 102 cloned candidates, which were associated with 30 outdated code snippets, appearing in 68 GitHub projects and manually investigated all of them. Out of the 102 cloned snippets, 13 cloned

snippets matched with themselves because some of the Qualitas projects also appear on GitHub. For other projects besides the Qualitas projects, 32 cloned snippets were not exactly the same (e.g., they contained additional code modifications made by the projects' developers or they were copied from another source with a slightly different code). 47 cloned snippets were the same as the outdated code snippets, which of 12 were buggy. Two cloned snippets gave attributions to Stack Overflow. The attributions pointed to different posts than the ones we found but containing the same code in the answers⁹. 32 cloned snippets were very likely to be a file-level clone from its respective original project (e.g., JFreeChart, JUnit, Log4J, Hadoop) based on their license header and the Javadoc comments. 13 cloned snippets did not have any hints or evidence of copying.

Interestingly, we discovered that the buggy version of the `humanReadableInt()` method from Hadoop appeared in two popular Java projects: deeplearning4j (8,830 stars and 4,223 forks) and Apache Hive (1,854 stars and 1,946 forks). Due to the lack of evidence, we could not conclude how this method, which is the same as the toxic code snippet we found on Stack Overflow, appears in the two projects. It is possible that the developers retrieved them from Stack Overflow, other websites, or from Hadoop code base directly. Nevertheless, we reported them to the developers of the two projects regarding the issue. We created a bug report for each project (deeplearning4j #4694¹⁰ and HIVE-18929¹¹) and communicated with the developers of the projects by describing the problem of race condition in the outdated version of the `humanReadableInt()` method and proposed a fix by using the newest version of the method in Hadoop. The issue has been fixed in both projects. The developers of deeplearning4j agreed that the method was problematic and decided to implement their own fix due to a concern of a potential software licensing conflict caused by copying the fix directly from the Hadoop code base. The Apache Hive developers investigated the code base and found that the `humanReadableInt()` method is not used anywhere in the project. Thus, they

⁹The answers were not marked as accepted so they were not included in our experiment.

¹⁰deeplearning4j bug report: <https://github.com/deeplearning4j/deeplearning4j/issues/4692>

¹¹Apache Hive bug report: <https://issues.apache.org/jira/browse/HIVE-18929>

deleted the method.

Although we did not find strong evidence of the outdated code snippets in GitHub projects, it would still be useful if Stack Overflow implements a flagging of outdated answers. The outdated online code clones cause problems ranging from uncompliable code (due to modifications and different API usage in the outdated code) to the introduction of vulnerabilities to the software [Xia et al., 2014]. An outdated code with a subtle change (e.g., Figure 4.1) may be copied and reused without awareness from developers. Moreover, an outdated code with a defect (e.g., a race condition problem in Figure 4.2) is clearly harmful to be reused. Although Stack Overflow has a voting mechanism that may mitigate this issue, the accepted answer may still be used by naive developers who copy and reuse the outdated code.

For RQ3, our results show that 65% (100) of QS clone pairs on Stack Overflow are outdated. 86 pairs differ from their newest versions by modifications applied to variable names or method names, added or deleted statements, to a fully rewritten code with new method signatures. 15 pairs are dead snippets. 47 outdated code snippets, of which 12 are buggy, are found in 130,719 GitHub projects without evidence of copying. A toxic code snippet with a race condition is found in two popular projects: deeplearning4j and Apache Hive.

4.4.4 RQ4: Software License Violations

Do license conflicts occur between Stack Overflow clones and their originals?

In our study, we reveal another type of toxic code snippets which is software licensing issues caused by code cloning to Stack Overflow. We found evidence that 153 pieces of code have been copied from Qualitas projects to Stack Overflow as examples. Another 109 pieces of code are cloned from external sources. Their status of accepted answers increases their chances of being reused. Even though most of the Qualitas projects came with a software license, we found that the license information was frequently missing after the code was copied to Stack Overflow. The licensing terms on top of source code files are not copied because usually only a small part of the file was cloned. In overall, we can see that most of the Stack

Table 4.13: License mapping of online clones (file-level)

Type	Qualitas	Stack Overflow (CC BY-NC-SA)	QS	EX	UD
Compatible	Apache-2	Apache-2	1		
	EPLv1	EPLv1	2		1
	Proprietary	Proprietary		2	
	Sun Microsystems	Sun Microsystems		3	
	No license	No license	20	9	2
	No license	CC BY-SA 3.0		1	
Total			23	15	3
Incompat.	AGPLv3/3+	No license	1		4
	Apache-2	No license	46	14	12
	BSD/BSD3	No license	4		1
	CDDL or GPLv2	No license		6	
	EPLv1	No license	10		6
	GPLv2+/3+	No license	8	48	7
	LesserGPLv2.1+/3+	No license	16		9
	MPLv1.1	No license		1	
	Oracle	No license		3	
	Proprietary	No license		1	2
	Sun Microsystems	No license		1	2
	Unknown	No license		11	
	LesserGPLv2.1+	New BSD3	1		
Total			86	78	50

Overflow snippets do not contain licensing terms while their clone counterparts in Qualitas projects and external sources do. Since licensing statement resides on top of a file, the results here are analysed at a file level, not clone fragment, and clone pairs from the same file are merged. The summary of licensing information is listed in Table 4.13.

Compatible license: There are 41 pairs which have compatible licenses such as *Apache license v.2*; *Eclipse Public License v.1 (EPLv1)*; or a pair of *Creative Common Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0)* vs. no license. These clones are safe for being reused. Since source code and text on Stack Overflow are protected by *CC BY-NC-SA 3.0*, we can treat the Stack Overflow code snippets without licensing information as having *CC BY-NC-SA 3.0* by default. The *CC BY-NC-SA 3.0* license is relaxed, and it only requests

an attribution when reused.

Incompatible license: there are 214 clone pairs which do not contain licensing information after they are posted on Stack Overflow or contain a different license from their Qualitas clone counterparts. Almost all (85) of **QS** clone pairs have their licensing terms removed or changed when posted on Stack Overflow. One QS clone pair posted by a JFreeChart developer changed its license from Lesser GPL v2.1+ to New BSD 3-clause. The developer may intentionally changed the license to be more suitable to Stack Overflow since New BSD 3-clause license allows reuse without requiring the same license in the distributing software or statement of changes.

For **EX** clone pairs, we searched for licensing terms of the original source code from the external sources. We found that 78 out of 93 EX clone pairs have incompatible licenses. Similarly, the license statement was removed from Stack Overflow snippets.

Of 53 **UD** clone pairs, 50 pairs have incompatible licenses. Again, most clones in Qualitas contain a license while the Stack Overflow snippets do not.

The same GitHub study has been done for license-incompatible code snippets. We detected clones between the 214 code snippets with their original license missing (86 QS, 78 EX, and 50 UD) and 130,719 GitHub projects using SourcererCC with 80% similarity threshold. Opposite to the outdated clones, we discovered a large number of 7,207 clone pairs. There were 95 pairs from 10 Qualitas projects hosted on GitHub and 7,112 pairs from 2,427 other projects. As shown in Table 4.14, the clones were found in highly-starred projects (29,465 to 10 stars) to 1-star projects. We found 12 code snippets with attributions to Stack Overflow questions/answers and 6 of them refer to one of our QS or EX clone pairs. We used the Ninka tool to identify software licenses of the 7,112 cloned code snippets automatically. Five code snippets did not have a license while the originals had the Apache-2, GPLv2, or GPLv1 license. One snippet had the AGPLv3 license while the original had the Apache-2 license. Only 995 code snippets in GitHub projects have the same license as the originals in Qualitas.

Note that the code snippets could potentially violate the license, but do not

No. of stars	Qualitas		Other Projects		
	Projects	Pairs	Projects	Pairs	Same license
29540 to 10	8	71	406	1,837	193
9 to 5	0	0	275	739	110
4 to 1	2	24	1,746	4,536	692
Total	10	95	2,427	7,112	995

Table 4.14: Clones of the 214 Stack Overflow missing-license code snippets in 130,719 GitHub projects

necessarily do so. In the example where the JFreeChart developer copied his own code, she or he was free to change the license. The same may have occurred with any of the 214 code snippets.

For RQ4, we found 214 code snippets on Stack Overflow that could potentially violate the license of their original software. The majority of them do not contain licensing statements after they have been copied to Stack Overflow. For 164 of them, we were able to identify, with evidence, where the code snippet has been copied from. We found occurrences of 7,112 clones of the 214 license-incompatible code snippets in 2,427 GitHub projects.

4.4.5 Overall Discussion

The findings lead to a few insights about online code clones and their toxicity as follows.

4.4.5.1 Online Code Clones Exists on Stack Overflow

In Chapter 3, the Stack Overflow answerers’ survey shows that the answerers sometimes copy code snippets from other sources, such as open source projects, to answer questions on Stack Overflow. The visitors’ survey shows that programmers reuse the code snippets in the answers and occasionally experience problems from the copied code. Our empirical study of clone detection between 72,365 Java code snippets on Stack Overflow and 111 open source projects in the curated Qualitas corpus support the survey results. We found 2,289 clone pairs reported by Simian and SourcererCC clone detectors and classified them using the seven patterns of online code cloning. We discovered 153 clone pairs that have been copied, with

evidence, from Qualitas projects to Stack Overflow, 109 clone pairs that have been copied from external sources besides Qualitas to Stack Overflow, and 65 clone pairs that are highly similar but without evidence of copying.

4.4.5.2 Outdated Clones Are Not Harmful

We found only a small number of toxic outdated code snippets in open source projects on GitHub. Besides 12 buggy and outdated code snippets found in 12 projects, the rest were non-harmful clones of the outdated code. Although other studies show that Stack Overflow code snippets may become toxic by containing security vulnerabilities [Acar et al., 2016, Fischer et al., 2017] or API misuse [Zhang et al., 2018], we found in this chapter that the damage caused by outdated code on Stack Overflow is not high.

4.4.5.3 License-incompatible Clones Can Be Harmful

The missing licensing statements of online code clones on Stack Overflow can cause more damage than the outdated code. As shown in our study and also in the study by An et al. [2017], some online clones on Stack Overflow are initially licensed under more restrictive license than Stack Overflow's CC BY-SA 3.0. If these missing-license online clones are reused in software with an incompatible license, the software owner may face legal issues. Software auditing services such as Black Duck Software or nexB, which can effectively check for license compliance of code copied from open source projects, do not check for the original license of the cloned code snippets on Stack Overflow. Although the Stack Overflow answerers who participated in our survey believe that most of the code snippets on Stack Overflow are too small to claim for copyright and they fall under fair-use, there is still a risk due to different legal systems in each country. For example, Germany's legal system does not have a concept of fair use. Besides, the number of minimum lines of code to be considered copying, i.e., de minimis, is also differently interpreted from case to case or from country to country.

4.4.5.4 Actionable Items

Our study discovers links from code in open source projects to code snippets on Stack Overflow using clone detection techniques. These links enable us to discover toxic code snippets with outdated code or licensing problems. The links can be exploited further to mitigate the problems of reusing outdated online clones and incompatible license on Stack Overflow code snippets. The thesis proposes the following actionable items:

Preventive measure: We encourage Stack Overflow to enforce attribution when source code snippets have been copied *from* licensed software projects to Stack Overflow. Moreover, an IDE plug-in that can automatically detect pasted source code and follow the link to Stack Overflow and then to the original open source projects could also prevent the issue of license violation. We foresee the implementation of the IDE plugin using a combination of scalable code clone detection [Sajnani et al., 2016] or clone search techniques [Kim et al., 2018] and automated software license detection [German et al., 2010]. In this chapter, we performed the check using a set of code clone detectors (Simian and SourcererCC) and software license detector (Ninka), but we had to operate the tools manually. Using the knowledge obtained from this chapter, we build an automated and scalable clone search with license detection as will be presented in Chapter 8. With the proposed solution, we demonstrate that the tool can create a database of code snippets on Stack Overflow and allow the users to search for clones and check their licenses. The clone search tool can offer a service via REST API and integrated into the IDE plugin. Every time a new code fragment is pasted into the IDE, the plugin performs the check by calling the clone search tool service and report the finding to the developers in real time.

Also, we also performed a study of two open source software auditing platforms/services: BlackDuck Software¹² and nexB¹³. For BlackDuck Software, we found from their report [BlackDuck CORSI, 2017] that while they check for code copied from open source projects including GitHub and Stack Overflow and analyse

¹²<https://www.blackducksoftware.com>

¹³<https://www.nexb.com>

their license compatibility with their customer software, the BlackDuck auditing system will treat the code snippets on Stack Overflow as having an “unknown” license because it does not know the original license of Stack Overflow code snippets. For nexB, their product does not mention checking of reused source code from Stack Overflow. So, our proposed service, which can offer more precise licensing information of Stack Overflow code snippets, will be useful as an add-on license check for code copying from Stack Overflow.

Detective measure: A system to detect outdated source code snippets on Stack Overflow may be needed. The system can leverage the online clone detection techniques in this chapter to periodically check if the cloned snippets are still up-to-date with their originals.

While checking if the copied code snippets on Stack Overflow are still up-to-date with the latest version of their originals can possibly be done automatically, it is a challenging task to automate the process of establishing the direction of code cloning. One viable solution, for now, is encouraging the Stack Overflow developers to always include the origin of the copied code snippet in the post so that this link is always established at the posting time. Even better, Stack Overflow can provide an optional form to fill in when an answerer post an answer if he or she copies the code from other software projects. The form should include the origin of the code snippet (possibly as a GitHub URL) and its original license. Using this manually established links at posting time, we can then automate the process of checking for an outdated code.

With such a system, the poster can be notified when the code has been updated in the original project so that he/she can update their code on Stack Overflow accordingly. On the other hand, with a crowdsourcing solution using an IDE plug-in, developers can also report the corrected version of outdated code back to the original Stack Overflow threads when they reuse outdated code and make corrections to them.

4.5 Threats to Validity

There are some potential threats to validity in this chapter. We separately discuss them in two aspects: internal and external validity.

4.5.1 Internal Validity:

We applied different mechanisms to ensure the validity of the clone pairs we classified. First, we used two widely-used clone detection tools, Simian and SourcererCC. We tried five other clone detectors but could not add them to the study due to their scalability issues and susceptibility to incomplete code snippets. We adopted Bellon's agreement metric [Bellon et al., 2007] to merge clone pairs for the manual classification and avoid double counting of the same clone pairs. We studied the impact of choosing different thresholds for Bellon's clone agreement and the minimum clone size of the two clone detectors and selected the optimal values. Nevertheless, our study might still suffer from false negatives, i.e., online code clones that are not reported by the tools or are filtered out by the size (less than 10 lines) during the clone detection process. We selected accepted answers on Stack Overflow in this chapter to focus on code snippets that solve the question's problem and are often shown on top of the answer list. We investigated the 72,365 Stack Overflow code snippets used in our study and found that 62,245 of them (86%) are also the highest voted answers.

Our seven patterns of online code cloning may not cover all possible online cloning patterns. However, instead of defining the patterns beforehand, we resorted to extracting them from the data sets. We derived them from a manual investigation of 679 online clone pairs and adopted one pattern from the study by Kapser and Godfrey [2003].

The 2,289 clone pairs classified by the two investigators are subject to manual judgement and human errors. Although we tried our best to be careful on searching for evidence and classifying the clones, some errors may still exist. We mitigated this problem by having two investigators to cross check the classifications and found 145 cases that lead to better classification results. This validation process can be even improved by employing an external investigator.

4.5.2 External validity:

We carefully chose the data sets for our experiment so the findings could be generalised as much as possible. We selected Stack Overflow because it is one of the most popular programming Q&A websites available with approximately 7.6 million users. There are a large number of code snippets reused from the site [An et al., 2017], and there are also several studies encouraging of doing so (e.g., Ponzanelli et al. [2013, 2014], Keivanloo et al. [2014], Park et al. [2014]). Nonetheless, it may not be representative to all the programming Q&A websites.

Regarding the code snippets, we downloaded a full data dump and extracted Java accepted answers since they are the most likely ones to be reused. Our findings are limited to these restrictions. They may not be generalised to all programming languages and all answers on Stack Overflow. We chose the curated Qualitas corpus for Java open source projects containing 111 projects [Tempero et al., 2010]. The projects span several areas of software and have been used in several empirical studies [Taube-Schock et al., 2011, Beckman et al., 2011, Vasilescu et al., 2011, Omar et al., 2012]. Although it is a curated and well-established corpus, it may not fully represent all Java open source software available. Lastly, we selected 130,719 GitHub Java projects based on the number of stars they obtained to represent their popularity. They might not represent all Java projects on GitHub, and the number of clone pairs found may differ from other project selection criteria.

4.6 Related Work

Work similar to ours are studies by An et al. [2017], Abdalkareem et al. [2017], Baltes et al. [2017], and Zhang et al. [2018]. An et al. investigated clones between 399 Android apps and Stack Overflow posts. They found 1,226 code snippets which were reused from 68 Android apps. They also observed that there are 1,279 cases of potential license violations. The authors rely on the timestamp to judge whether the code has been copied from/to Stack Overflow along with confirmations from six developers. Instead of Android apps, we investigated clones between Stack Overflow and 111 open source projects. Their results are similar to

our findings that there exist clones from software projects to Stack Overflow with potential license violations. Abdalkareem et al. [2017] detected clones between Stack Overflow posts and Android apps from the F-Droid repository and used timestamps to determine the direction of copying. They found 22 apps containing code cloned from Stack Overflow. They reported that cloned code is commonly used for enhancing existing code. Their analysis shows that the cloned code from Stack Overflow has detrimental effects on quality of the apps. Baltes et al. [2017] discovered that two-thirds of the clones from the 10 most frequently referenced Java code snippets on Stack Overflow do not contain attributions. Zhang et al. [2018] study quality of code snippets on Stack Overflow. They show that 31% of the analysed Stack Overflow posts contain potential API usage violations and could lead to program crashes or resource leaks.

4.7 Chapter Summary

The findings in this chapter establish the existence of online code clones and their potential ramifications. This chapter provides an incentive for creating a scalable clone search engine to effectively detect clones that are originated from online sources. Since we target online sources such as Stack Overflow or GitHub, the clone search engine can build and keep a large database of online code fragments, which can be queried multiple times.

In order to develop such a tool, we take a step back to look at what has already been invented by investigating the strengths and weaknesses of the current state-of-the-art code similarity techniques. The next chapter will discuss a framework for evaluating code similarity and clone search tool, called OCD. We will explain how the framework is created and present the results of using the framework to compare 34 code similarity analysers. Lastly, we will discuss how the results from the study affects our design of the clone search tool.

Chapter 5

OCD: A Framework for Evaluating Code Similarity and Clone Search Tools

This chapter explains a framework for evaluating code similarity and a code clone search tool called **OCD** (Obfuscation/Compilation/Decompilation) and presents the results of comparing 34 state-of-the-art code similarity analysers using the framework. We built the framework as a benchmark for evaluating not only code clone detectors but various types of code similarity detection tools. Moreover, the framework supports the thesis's goal of creating a scalable code clone search tool by allowing the author to learn the strengths and weaknesses of the state-of-the-art tools and wisely choose an appropriate method for large-scale code similarity measure.

This chapter sets off by explaining the OCD framework. Then, the chapter puts the framework to use by performing an empirical study to compare 34 code similarity analysers on pervasively modified source code and boiler-plate code data sets and studying the sensitivity of the tools' configurations to the data sets. The chapter also applies compilation/decompilation as a code normalisation method and evaluates their effects to the tools' performance. Lastly, the chapter discusses lessons learned from the study and how the results influence our design of a scalable code clone search technique.

5.1 Motivation

The assessment of source code similarity has a co-evolutionary relationship with the modifications made to the code at the point of its creation. Although there is a large number of clone detectors, plagiarism detectors, and code similarity detectors invented in the research community, there are relatively few studies that compare and evaluate their performances. Burd and Bailey [2002] compare five clone detectors for preventive maintenance tasks. Bellon et al. [2007] created and applied a framework for comparing and evaluating 6 clone detectors. Roy et al. [2009] evaluated a large set of clone detection tools but only based on results obtained from the tools' published papers. Hage et al. [2010] compare five plagiarism detectors against 17 code modifications. Biegel et al. [2011] compare three code similarity measures to identify code that needs refactoring. Svajlenko and Roy [2016] developed and used a clone evaluation framework called BigCloneEval to evaluate 10 state-of-the-art clone detectors. Although these studies cover various goals of tool evaluation and cover different types of code modification found in the chosen data sets, they suffer from two limitations: (1) the selected tools are limited to only a subset of clone or plagiarism detectors, and (2) the results are based on different data sets, so one cannot compare a tool's performance from one study to another tool's from another study. To the best of our knowledge, there is no study that performs a comprehensive and fair comparison of widely-used code similarity analysers based on the same data sets.

In this chapter, we fill the gap by presenting a framework for comparing code similarity analysers and use it to do the largest extant study on source code similarity that covers the widest range of techniques and tools. We study the tools' performances on both local and pervasive (global) code modifications usually found in software engineering activities such as code cloning, software plagiarism, and code refactoring. This study is motivated by the question:

“When source code is copied and modified, which code similarity detection techniques or tools get the most accurate results?”

To answer this question, we use our framework to evaluate the performance

of the current state-of-the-art similarity detection techniques using several error measures. The aim of this study is to provide a foundation for the appropriate choice of a similarity detection technique or tool for a given application based on a thorough evaluation of strengths and weaknesses on source code with local and global modifications. Choosing the wrong technique or tool with which to measure software similarity or even just choosing the wrong parameters may have detrimental consequences. The framework can also be used for evaluating new code similarity tools and compare the performance to our reported results of the current tools.

For the empirical study, we have selected as many techniques for source code similarity measurement as possible, 34 in all, covering techniques specifically designed for clone and plagiarism detection, plus the normalised compression distance, string matching, and information retrieval. In general, the selected tools require the optimisation of their parameters as these can affect the tools' execution behaviours and consequently their results. A previous study regarding parameter optimisation [Wang et al., 2013b] has explored only a small set of clone detectors' parameters using search-based techniques. Therefore, while including more tools in this study, we have also searched through a wider range of configurations for each tool, studied their impact, and discovered the best configurations for each data set in our experiments. After obtaining tools' optimal configurations derived from one data set, we apply them to another data set and observe if they can be reused effectively.

Clone and plagiarism detection use intermediate representations like token streams or abstract syntax trees or other transformations like pretty printing or comment removal to achieve a normalised representation [Roy et al., 2009]. We integrated compilation and decompilation as a normalisation pre-process step for similarity detection and evaluated its effectiveness.

5.2 Contributions

This chapter makes the following primary contributions:

1. **A framework and a data set of pervasive code modifications:** Our OCD framework is built for comparing code similarity analysers based on a general similarity report template. It aims to evaluate code similarity detection tools on source code with pervasive modifications, which is a challenging scenario for code similarity. The similarity report template is designed to support the evaluation using either pair-based or query-based measures. Thus, it is suitable for both code clone/plagiarism detection and clone search tool evaluation. The generated Java data set with pervasive modifications used in this study has been created to be challenging for code similarity analysers. According to the way we constructed the data set, the complete ground truth is known. We make the data set publicly available so that it can be used in future studies of tool evaluation and comparison.
2. **A broad, thorough study of the performance of similarity tools and techniques:** Using our framework, we compare a large range of 34 similarity detection techniques and tools using five experimental scenarios for Java source code in order to measure the techniques' performances and observe their behaviours. We apply several error measures including pair-based and query-based measures. The results show that, in overall, highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures. However, we also observed some situations where compression-based and textual similarity tools are recommended over clone and plagiarism detectors.

The results of the evaluation can be used by researchers as guidelines for selecting techniques and tools appropriate for their problem domain. Our study confirms both that tool configurations have strong effects on tool performance and that they are sensitive to particular data sets. Poorly chosen techniques or configurations can severely affect results.
3. **Normalisation by decompilation:** Our study confirms that compilation and decompilation as a pre-processing step can normalise pervasively modified

source code and can improve the effectiveness of similarity measurement techniques with statistical significance. Three of the similarity detection techniques and tools reported no false classifications once such normalisation was applied.

5.3 Background

5.3.1 Source Code Modifications

We are interested in two scenarios of code modifications in this chapter: pervasive code modifications (global) and boiler-plate code (local). Their definitions are as follows.

5.3.1.1 Pervasive Modifications

Pervasive modifications are code changes that affect the code globally across the whole file with multiple changes applied one after another. These are code transformations that are mainly found in the course of software plagiarism when one wants to conceal copied code by changing their appearance and avoid detection [Daniela et al., 2012]. Nevertheless, they also represent code clones that are repeatedly modified over time during software evolution [Pate et al., 2013], and source code before and after refactoring activities [Fowler, 2013]. However, our definition of pervasive modifications excludes strong obfuscation [Collberg et al., 1997], that aims to protect code from reverse engineering by making it difficult or impossible to understand.

Most clone or plagiarism detection tools and techniques tolerate different degrees of change and still identify cloned or plagiarised fragments. However, while they usually have no problem in the presence of local or confined modifications, pervasive modifications that transform whole files remain a challenge [Roy and Cordy, 2009a], for example, in a situation that multiple methods are merged into a single method due to a code refactoring activity. A clone detector focusing on method-level clones would not report the code before and after merging as a clone pair. Moreover, with multiple lexical and structural code changes applied repeatedly at the same time, resulting source code can be totally different. When one looks at

```

/* original */
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo, j = hi+1;
    Comparable v = a[lo];
    while (true) {
        while (less(a[++i], v)) if (i == hi) break;
        while (less(v, a[--j])) if (j == lo) break;
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}

/* plagiarised code */
private static int partition(int[] bob, int left, int right) {
    int x = left;
    int y = right+1;
    for (;;) {
        while (less(bob[left], bob[--y]))
            if (y == left) break;
        while (less(bob[++x], bob[left]))
            if (x == right) break;
        if (x >= y) break;
        swap(bob, y, x);
    }
    swap(bob, y, left);
    return y;
}

```

Figure 5.1: Pervasive modifications found in a programming submission.

code before and after applying pervasive modifications, one might not be able to tell that both originate from the same file. We found that code similarity detection tools have the same confusion.

We define source code with pervasive modifications to contain a combination of the following code changes:

1. Lexical changes of formatting, layout modifications (Type I clones), and identifier renaming (Type II clones).
2. Structural changes, e.g., `if` to `case` or `while` to `for`, or insertions or deletions of statements (Type III clones).
3. Extreme code transformations that preserve source code semantics but change its syntax (Type IV clones).

```

private void createConnectionThread(int input) {
    data = new HoldSharedData(startTime, password, pwdCounter);
    int numThreads = input;
    int batch = pwdCounter/numThreads + 1;
    numThreads = pwdCounter/batch + 1;
    System.out.println("Number of Connection Threads Used=" +
        numThreads);
    ConnectionThread[] connThread = new
        ConnectionThread[numThreads];
    for(int index = 0; index < numThreads; index++) {
        connThread[index] = new ConnectionThread(url, index, batch,
            data);
        connThread[index].conn();
    }
}

```

Figure 5.2: A boiler-plate code to create connection threads.

Figure 5.1 shows an example of code before and after applying pervasive modifications. It is a real-world example of plagiarism from a university’s programming class submission¹.

5.3.1.2 Boiler-plate Code

Boiler-plate code occurs when developers reuse a code template, usually a function or a code block, to achieve a particular task. It has been defined as one of the code cloning patterns by Kapser and Godfrey [2006, 2008]. Boiler-plate code can be found when building device drivers for operating systems [Baxter et al., 1998], developing android applications [Crussell et al., 2013], and giving programming assignments [Burrows et al., 2007, Schleimer et al., 2003]. Boiler-plate code usually contains small code modifications in order to adapt the boiler-plate code to a new environment. In contrast to pervasive modifications, the modifications made to boiler-plate code are usually contained in a function or block. Figure 5.2 depicts an example of boiler-plate code used for creating new HTTP connection threads which can be reused as-is or with minimum changes.

5.3.2 Obfuscation and Deobfuscation

Obfuscation is a mechanism of making changes to a program while preserving its original functions. It originally aimed to protect intellectual property of computer

¹<https://www.princeton.edu/pr/pub/integrity/pages/plagiarism/>

programs from reverse engineering or from malicious attack [Collberg et al., 2002] and can be achieved in both source and binary level. Many automatic code obfuscation tools are available nowadays both for commercial (e.g., Semantic Designs Inc.’s C obfuscator [Designs, 2015], Stunnix’s obfuscators [Stunnix, 2015], Diablo [PARIS research group, 2015]) and research purposes [Chow et al., 2001, Schulze and Meyer, 2013, Madou et al., 2006, Necula et al., 2002].

Given a program P , and the transformed program P' , the definition of obfuscation transformations T is $P \xrightarrow{T} P'$ requiring P and P' to hold the same observational behaviour². Specifically, *legal* obfuscation transformation requires: 1) if P fails to terminate or terminates with errors then P' may or may not terminate, and 2) P' must terminate if P terminates [Collberg et al., 1997].

Generally, there are three approaches for obfuscation transformations: lexical (layout), control, and data transformation [Collberg et al., 2002, 1997]. Lexical transformations can be achieved by renaming identifiers and formatting changes, while control transformations use more sophisticated methods such as embedding spurious branches and opaque predicates which can be deducted only at runtime. Data transformations make changes to data structures and hence make the source code difficult to reverse engineer. Similarly, binary-code obfuscators transform the content of executable files.

Many obfuscation techniques have been invented and put to use in commercial obfuscators. Collberg et al. [2003] introduce several reordering techniques (e.g., method parameters, basic block instructions, variables, and constants), splitting of classes, basic blocks, and arrays, and merging of methods, parameters, and classes. These techniques are implemented in their tool, SandMark. Wang et al. [2001] propose a sophisticated deep obfuscation method called *control flow flattening* which is used in a commercial tool called Cloakware. ProGuard [Guard Square, 2015] is a Java bytecode obfuscator which performs obfuscation by removing existing names³ (e.g., class, method names), replacing them with meaningless

²Observation behaviour is loosely defined by Collberg et al. [1997] as “behavior as experienced by the user.” Thus, side-effects of P' that P does not have (e.g., file creation or message transmission over the network), which are not experienced by the user, are not taken into account.

³This renaming is not applied to names of external library entities, which cannot be replaced.

characters, and also gets rid of all debugging information from Java bytecode. Loco [Madou et al., 2006] is a binary obfuscator capable of performing obfuscation using control flow flattening and opaque predicates on selected fragments of code.

Deobfuscation is a method aiming at reversing the effects of obfuscation which can be achieved at either static and dynamic level. It can be useful in many aspects such as detection of obfuscated malware [Nachenberg, 1996] or as a resiliency test for a newly developed obfuscation method [Madou et al., 2006]. While *surface obfuscation* such as variable renaming can be handled straightforwardly, *deep obfuscation* which makes large changes to the structure of the program (e.g., opaque predicates or control flow flattening) is much more difficult to reverse. However, it is not totally impossible. It has been shown that one can counter control flow flattening by either cloning the portions of added spurious code to separate them from the original execution path or use static path feasibility analysis [Udupa et al., 2005].

5.3.3 Program Decompilation

Decompilation of a program generates high-level code from low-level code. It has several benefits including recovery of lost source code from compiled artifacts such as binary or bytecode, reverse engineering, finding similar applications [Chen et al., 2014]. On the other hand, decompilation can also be used to create program clones by decompiling a program, making changes, and repacking it into a new program. An example of this malicious use of decompilation can be seen from a study by Chen et al. [2014]. They found that 13.51% of all applications from five different Android markets are clones. Gibler et al. [2013] discovered that these decompiled and cloned apps can divert advertisement impressions from the original app owners by 14% and divert potential users by 10%.

Many decompilers have been invented in the literature for various programming languages [Cifuentes and Gough, 1995, Proebsting and Watterson, 1997, Desnos and Gueguen, 2011, Mycroft, 1999, Breuer and Bowen, 1994]. Several techniques are involved to successfully decompile a program. The decompiled source code may be different according to each particular decompiler. Conceptually,

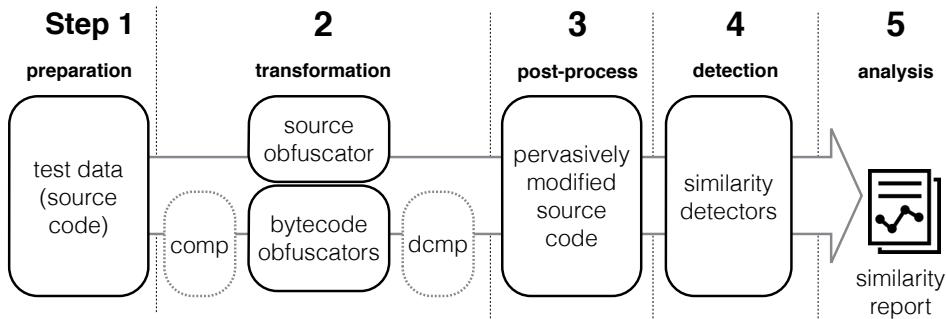


Figure 5.3: The OCD framework

decompilers extract semantics of programs from their executables, then, with some heuristics, generate the source code based on this extraction. For example Krakatoa [Proebsting and Watterson, 1997], a Java decompiler, extracts expressions and type information from Java bytecode using symbolic execution, and creates a control flow graph (CFG) of the program representing the behaviour of the executable. Then, to generate source code, a sequencer arranges the nodes and creates an abstract syntax tree (AST) of the program. The AST is then simplified by rewriting rules and, finally, the resulting Java source code is created by traversing the AST.

5.4 The OCD Framework

In this chapter, we introduce a framework called the *OCD (Obfuscation/Compilation/Decompilation) framework*. The overview of the framework is shown in Figure 5.3, which consists of 5 main steps.

In Step 1, the test data, i.e., a collection of Java source code files, are created. In Step 2, pervasively modified variants (clones) are generated by applying code transformations on the original test data files. The generated variants are saved to a new file. Multiple tools can be chosen to transform the source code in this step to generate different combinations of the variants. For example, if we have an original file F and we select one source code obfuscator O and one decompiler D , we retrieve three variants of F including F_O , F_D , and F_{OD} . In Step 3, the original and the variants are normalised. A simple form of normalisation is pretty printing by removing comments and formatting the code to a specific coding convention,

which is usually applied to source code during clone detection [Roy and Cordy, 2008]. The framework user can also choose to add his or her own normalisation technique in this step. In Step 4, a code similarity detection tool that the user wants to evaluate is plugged into the framework. The tool must be executed on every file pair in the data set to generate a similarity report containing similarity values for all the pairs.

In Step 5, the similarity report is analysed. We extract a similarity value $\text{sim}(x, y)$ from the report for every pair of files (x, y) , and classify the pair as being similar (clones) or not similar based on a chosen threshold T . The set $\text{Sim}(F)$ of similar pairs out of all the file pairs in F is

$$\text{Sim}(F) = \{(x, y) \in F \times F : \text{sim}(x, y) > T\} \quad (5.1)$$

According to the way we generate the data set, we obtain a complete ground truth, i.e., we know exactly which pair is similar and which pair is not. The files that are originated from the same Java file must be treated as similar, and vice versa. Thus, we can decide whether a code pair is correctly classified as a similar pair (true positive, TP), correctly classified as a dissimilar pair (true negative, TN), incorrectly classified as a similar pair while it is actually dissimilar (false positive, FP), and incorrectly classified as dissimilar pair while it is actually a similar pair (false negative, FN). After the classification, we create a confusion matrix for the tool containing the number of TP , FP , TN , and FN . Computation of precision, recall, accuracy, or F1 score is then based on the confusion matrix.

5.4.1 The Similarity Report Template

We design a template for the similarity report, which is produced after executing a code similarity detection tool on our framework. The report is a Comma-Separated Values (CSV) file that contains a matrix of similarity values of all the pairwise comparisons. The matrix can be either symmetric or asymmetric depending on the code similarity detection tool. It is formatted as follows.

1. The first row (header) contains a list of file names in the data set separated by

-	A.java	B.java	C.java	D.java	E.java	F.java	G.java	H.java	I.java	J.java
A.java	100	55	37	63	33	43	35	60	32	43
B.java	55	100	35	54	33	39	36	56	32	39
C.java	37	35	100	39	60	27	80	36	58	27
D.java	63	54	39	100	34	58	37	80	33	58
E.java	33	33	60	34	100	34	60	33	82	34
F.java	43	39	27	58	34	100	26	59	34	100
G.java	35	36	80	37	60	26	100	36	58	26
H.java	60	56	36	80	33	59	36	100	32	59
I.java	32	32	58	33	82	34	58	32	100	34
J.java	43	39	27	58	34	100	26	59	34	100

Figure 5.4: An example similarity report

commas.

2. The first column in each row contains the name of the file being compared, followed by a list of similarity values between the file and the other files separated by a comma.
3. The report must have identical row and column sizes, i.e., it resembles an $N \times N$ matrix, where N is the number of files in the data set.

An example similarity report is illustrated in Figure 5.4. Let assume the dataset contains 10 files: A.java, B.java, C.java, ..., J.java. Thus, the report has a dimension of 10×10 . The first row contains a list of the file names in the data set while the rest of the rows contains the similarity values between the pair of the file in the first column and the file in the other columns. For example, the value 35 at row 2 column 3, i.e., [2, 3], is a comparison [B.java, C.java]. The same value of 35 is found at [3, 2] since it is the reverse comparison [C.java, B.java]. We keep the similarity scores for both directions because some code similarity analysers do not give symmetric similarity values. Thus, [B.java, C.java] does not always equal [C.java, B.java].

The similarity report contains only file names and similarity values. Hence, it offers a benefit of supporting any tool and technique as long as the similarity values can be obtained or derived from the tool's output. For example, in the empirical study shown later in this chapter, we employed a method to convert clone pair information in a code clone report to a similarity score based on a ratio of number

of cloned lines.

5.4.2 Implementation of the Framework

We implement the OCD framework by using a combination of various open source programs and our developed programs. Mainly, it covers the following sets of tools: (1) the tools to generate pervasive code modifications and perform code normalisation, (2) the tools to run code similarity analysers, and (3) the tools to read the similarity report and output error measure scores.

5.4.2.1 Pervasive Code Modification and Code Normalisation Tools

We deploy two types of tools for creating pervasively modified source code: code obfuscators and compiler/decompilers. For code normalisation, we rely on a Java pretty-printing tool and decompilers.

Obfuscators: In order to create pervasively modified source code variants in

Table 5.1: List of pervasive code modifications offered by our source code and bytecode obfuscator, and compiler/decompilers

Code modifications	Artifice	ProGuard	(De)compilers
<i>Lexical modification</i>			
Formatting changes [Roy and Cordy, 2009a, Duric and Gasevic, 2013, Joy and Luck, 1999]	✓		✓
Addition, modification or deletion of comments [Duric and Gasevic, 2013, Joy and Luck, 1999]	✓		✓
Renaming of identifiers, methods [Roy and Cordy, 2009a, Duric and Gasevic, 2013, Joy and Luck, 1999, Brixtel et al., 2010, Fowler, 2013]	✓	✓	✓
Modification of constant values [Duric and Gasevic, 2013]		✓	
<i>Structural modification</i>			
Split or merge of variable declarations [Duric and Gasevic, 2013]	✓		✓
Addition, modification or deletion of modifiers [Duric and Gasevic, 2013, Fowler, 2013]		✓	✓
Line insertion/deletion with further edits [Roy and Cordy, 2009a]	✓		✓
Reordering of statements & control replacements [Roy and Cordy, 2009a, Duric and Gasevic, 2013, Joy and Luck, 1999, Brixtel et al., 2010]	✓	✓	✓
Modification of control structures [Duric and Gasevic, 2013, Joy and Luck, 1999, Brixtel et al., 2010]	✓		✓
Changing of data types and modification of data structures [Duric and Gasevic, 2013]			✓
Method inlining and method refactoring [Duric and Gasevic, 2013, Fowler, 2013]		✓	
Structural redesign of source code [Duric and Gasevic, 2013, Fowler, 2013]			✓

Step 2 of the framework, we employed two obfuscators: Artifice and ProGuard. Artifice [Schulze and Meyer, 2013] is an Eclipse plugin for source-level obfuscation. The tool makes 5 different transformations to Java source code including 1) renaming of variables, fields, and methods, 2) changing assignment, increment, and decrement operations to normal form, 3) inserting additional assignment, increment, and decrement operations when possible, 4) changing `while` to `for` and the other way around, and 5) changing `if` to its short form. We manually ran Artifice in Eclipse.

ProGuard [Guard Square, 2015] is a well known open-source bytecode obfuscator. It is a versatile tool containing several functions including shrinking Java class files, optimisation, obfuscation, and pre-verification. ProGuard obfuscates Java bytecode by renaming classes, fields, and variables with short and meaningless ones. It also performs package hierarchy flattening, class repackaging, merging methods/classes and modifying package access permissions.

Using source and bytecode obfuscators, we can create pervasively modified source code that contains modifications of lexical and structural changes. We have investigated the code transformations offered by Artifice and ProGuard and found that they cover changes commonly found in both code cloning and code plagiarism as reported by [Roy and Cordy, 2009a, Schulze and Meyer, 2013, Duric and Gasevic, 2013, Joy and Luck, 1999, Brixel et al., 2010]. The details of code modifications supported by our obfuscators are shown in Table 5.1.

Compiler and Decompilers: One can use a combination of compilation and decompilation as a method of source code obfuscation or transformation. Luo et al. [2014] use GCC/G++ with different optimisation options to generate 10 different binary versions of the same program. However, if the desired final product is source code, a decompiler is also required in the process in order to transform the bytecode back to its source form. The only compiler deployed in this study is the standard Java compiler (`javac`).

Decompilation is a method for reversing the process of program compilation. Given a low-level language program such as an executable file, a decompiler gen-

erates a high-level language counterpart that resembles the (original) source code. This has several applications including recovery of lost source code, migrating a system to another platform, upgrading an old program into a newer programming language, restructuring poorly-written code, finding bugs or malicious code in binary programs, and program validation [Cifuentes and Gough, 1995]. An example of using the decompiler to reuse code is a well-known lawsuit between Oracle and Google [Oracle America, Inc. v. Google Inc., 2011]. It seems that Google decompiled a Java library to obtain the source code of its APIs and then partially reused them in their Android operating system.

Since each decompiler has its own decompiling algorithm, one decompiler usually generates source code which is different from the source code generated by other decompilers. Using more than one decompiler can also be a method of obfuscation by creating variants of the same program with the same semantics but with different source code.

We selected two open source decompilers: Krakatau and Procyon. Krakatau [Grosse, 2016] is an open-source toolset comprising a decompiler, a class file disassembler, and an assembler. Procyon [Strobel, 2015] includes a Java open-source decompiler. It has advantages over other decompilers for declaration of `enum`, `String`, `switch` statements, anonymous and named local classes, annotations, and method references. They are used in both the transformation (obfuscation) and normalisation post-process steps (Steps 2 and 3) of the framework.

Using a combination of compilation and decompilation to generate code with pervasive modifications can represent source code that has been refactored [Fowler, 2013] or rewritten (i.e., Type-4 clones) [Roy et al., 2009]. While its semantics has been preserved, the source code syntax including layout, variable names, and structure may be different. Table 5.1 shows code modifications that are supported by our compiler and decompilers.

Code Normalisation: After the source code has been pervasively modified by the obfuscators, compiler, and decompilers, we applied `astyle`⁴ pretty-printing

⁴<http://astyle.sourceforge.net>

tool to reformat the code to follow the same Java coding convention. This is important since some general string similarity techniques are sensitive to whitespace, indentations, or newlines. Then, we remove comments from the source code using `uncomment` tool⁵. This is important due to the sensitivity of string-based measures on code comments and also due to extra comments generated by some tools such as the Procyon decompiler after decompilation. Any additional code normalisation techniques can be added at this step. In our empirical study, we also performed another code normalisation operation by applying the compiler/decompilers on the pervasively modified source code one more time.

5.4.2.2 Tools for Running Code Similarity Analysers

To generate a similarity report according to our predefined template, we have to correctly manage the order of pairwise comparisons, execute the tool with specified parameters, and collect the tool’s results and format them. We rely on a bash script to perform this task. The script is responsible for managing the tool’s dependencies and executing the tool on a given source code pair. The script then collects the tool result; which can be a similarity score, a terminal output, or a report file; derives a similarity score, and writes the score to the similarity report file. An example tool running script can be found in Appendix C. The framework user has to adapt the script to work with their own tool.

5.4.2.3 Tools to Analyse the Similarity Report

We create another set of scripts to read the similarity report and generate error measure scores. The scripts support computation of pair-based measures including precision, recall, accuracy, and F1 scores and query-based measures including precision at n (`prec@n`), average R-precision (ARP) and mean average precision (MAP) scores. The script is also responsible for finding the best similarity cut-off threshold that gives the highest F1, and `prec@n`.

⁵The `uncomment` tool is created by Kimmo Kulovesi (<http://arkku.com>)

5.4.3 Using the OCD Framework

To use the OCD framework to evaluate a code similarity analyser. The user should follow the guideline in Appendix C. The framework comes with an OCD data set, and the scripts to run the tool and analyse the result. The user can choose to execute the tool on the OCD data set and obtain the benefit of comparing his or her results to our results of 34 code similarity analysers reported in this chapter or plugging in their custom data set for a specific need.

5.5 Empirical Study

We have performed an empirical study of the current state-of-the-art code similarity detection techniques using the OCD framework. Our empirical study consists of five experimental scenarios covering different aspects and characteristics of source code similarity. Three experimental scenarios examined the tool/technique performance on three different data sets to discover any strengths and weaknesses. These three are (1) an experiment on the products of the two obfuscation tools, (2) an experiment on an available data set for identification of reuse boiler-plate code [Flores et al., 2014], and (3) an experiment on the combinations of pervasive modifications and boiler-plate code. The fourth scenario examined the effectiveness of compilation/decompilation as a preprocessing normalisation strategy and the fifth evaluated the use of error measures from information retrieval for comparing tool performance without relying on a threshold value.

5.5.1 Research Questions

The empirical study aimed to answer the following research questions:

RQ1 (Performance comparison): *How well do current similarity detection techniques perform in the presence of pervasive source code modifications and boiler-plate code?* We compare 34 code similarity analysers using a data set of 100 pervasively modified pieces of source code and a data set of 259 pieces of Java source code that incorporate reused boiler-plate code.

RQ2 (Optimal configurations): *What are the best parameter settings and*

similarity thresholds for the techniques? We exhaustively search wide ranges of the tools’ parameter values to locate the ones that give optimal performances so that we can fairly compare the techniques. We are also interested to see if one can gain optimal performance of the tools by relying on default configurations.

RQ3 (Normalisation by decompilation): *How much does compilation followed by decompilation as a pre-processing normalisation method improve detection results for pervasively modified code?* We apply compilation and decompilation to the data set before running the tools. We compare the performances before and after applying this normalisation.

RQ4 (Reuse of configurations): *Can we effectively reuse optimal tool configurations for one data set on another data set?* We apply the optimal tool configurations obtained using one data set when using the tools with another data set and investigate whether they still offer the tools’ best performances.

RQ5 (Ranked Results): *Which tools perform best when only the top n results are retrieved?* Besides the set-based error measures normally used in clone and plagiarism detection evaluation (e.g., precision, recall, F-scores), we also compare and report the tools’ performances using ranked results adopted from information retrieval. This comparison has a practical benefit in terms of plagiarism detection, manual clone investigation, and automated software repair.

RQ6 (Pervasive Modifications + Boiler-plate Code): *How well do the techniques perform when source code containing boiler-plate code clones has been pervasively modified?* We evaluate the tools on a data set combining both local and global code modifications. This question also studies which types of pervasive modifications (source code obfuscation, bytecode obfuscation, compilation/decompilation) strongly affect tools’ performances.

5.5.2 Code Similarity Detection Tools and Techniques

The code similarity detectors cover a wide range of similarity measurement techniques and methods including plagiarism and clone detection, compression distance, string matching, and information retrieval. All tools are open source in order to expedite the repeatability of our experiments.

5.5.2.1 Plagiarism Detectors

The selected plagiarism detectors include JPlag, Sherlock, Sim, and Plaggie. JPlag [Prechelt et al., 2002] and Sim [Gitchell and Tran, 1999] are token-based tools which come in versions for text (jplag-text and simtext) and Java (jplag-java and simjava), while Sherlock [Pike and Loki, 2002] relies on digital signatures, i.e., a number created from a series of bits converted from the source code text. Plaggie's detection [Ahtiainen et al., 2006] method is not public but claims to have the same functionalities as JPlag. Although there are several other plagiarism detection tools available, some of them could not be chosen for the study due to the absence of command-line versions preventing them from being automated. Moreover, we require a quantitative similarity measurement so we can compare their performances. All chosen tools report a numerical similarity value, $\text{sim}(x,y)$, for a given file pair x,y .

5.5.2.2 Clone Detectors

We cover a wide spectrum of clone detection techniques including text-based, token-based, and tree-based techniques. Like the plagiarism detectors, the selected tools are command-line based. However, most state-of-the-art clone detectors do not report a similarity value between two files. Thus, we adopted the *General Clone Format (GCF)* as a common format for clone reports. We modified and integrated the *GCF Converter* [Wang et al., 2013b] to convert clone reports generated by unsupported clone detectors into GCF format.

Since a GCF report contains several clone fragments found between two files x and y , the similarity of x to y can be calculated as the ratio of the size of clone fragment between x and y found in x (overlaps are handled), i.e., $\text{frag}_i^x(x,y)$, to the

size of x and vice versa.

$$\text{simGCF}(x, y) = \frac{\sum_{i=1}^n |\text{frag}_i^x(x, y)|}{|x|} \quad (5.2)$$

Using this method, we included six state-of-the-art clone detectors: CCFinderX, NiCad, Simian, iClones, Deckard, and SourcererCC. CCFinderX (ccfx) [Kamiya et al., 2002] is a token-based clone detector detecting similarity using suffix trees. NiCad [Roy and Cordy, 2008] is a clone detection tool embedding TXL for pretty-printing, and compares source code using string similarity. Simian [Harris, 2003] is a purely text-based clone detection tool relying on text line comparison with a capability for checking basic code modifications, e.g., identifier renaming. iClones [Göde and Koschke, 2009] performs token-based incremental clone detection over several revisions of a program. Deckard [Jiang et al., 2007a] converts source code into an AST and computes similarity by comparing characteristic vectors generated from the AST to find cloned code based on approximate tree similarity. SourcererCC [Sajnani et al., 2016] is a scalable token-based clone detection tool using an optimised inverted index with two token filtering heuristics for fast clone detection.

Although most of the clone reports only contain clone lines, the actual implementation of clone detection tools works at a different granularity of code fragments. Measuring clone similarity at a single granularity level, such as line, may penalise some tools while favouring another set of tools. With this concern in mind, our clone similarity calculation varies over multiple granularity levels to avoid biases to any particular tools. We consider three different granularity levels: line, token, and character. Computing similarity at a level of lines or tokens is common for clone detectors. Simian and NiCad detect clones based on source code lines while CCFinderX, iClones, and SourcererCC work at token level. However, Deckard compares clones based on ASTs so its similarity comes from neither lines nor tokens. To make sure that we get the most accurate similarity calculation for Deckard and other clone detectors, we also cover the most fine-grained level of source code: characters. Using these three levels of granularity (line, word, and

character), we calculate three $\text{sim}_{\text{GCF}}(x, y)$ values for each of the tools.

5.5.2.3 Compression Tools

Normalised compression distance (NCD) is a distance metric between two documents based on compression [Cilibrasi and Vitányi, 2005]. It is an approximation of the normalised information distance which is in turn based on the concept of Kolmogorov complexity [Li and Vitányi, 2008]. The NCD between two documents can be computed by

$$\text{NCD}_Z(x, y) = \frac{Z(xy) - \min\{Z(x), Z(y)\}}{\max\{Z(x), Z(y)\}} \quad (5.3)$$

where $Z(x)$ means the length of the compressed version of document x using compressor Z . In this study, five variations of NCD tools are chosen. One is part of CompLearn [Cilibrasi et al., 2015] which uses the built-in bzlib and zlib compressors. The other four have been created by the authors as shell scripts. The first one utilises 7-Zip [Pavlov, 2016] with various compression methods including BZip2, Deflate, Deflate64, PPMd, LZMA, and LZMA2. The other three rely on Unix's gzip, bzip2, and xz compressors respectively.

Lastly, we define another, asymmetric, similarity measurement based on compression called *inclusion compression divergence (ICD)*. It is a compressor based approximation to the ratio between the conditional Kolmogorov complexity of string x given string y and the Kolmogorov complexity of x , i.e., to $K(x|y)/K(x)$, the proportion of the randomness in x not due to that of y . It is defined as

$$\text{ICD}_Z(x, y) = \frac{Z(xy) - Z(y)}{Z(x)} \quad (5.4)$$

and when C is NCD_Z or ICD_Z then we use $\text{sim}_C(x, y) = 1 - C(x, y)$.

5.5.2.4 Other Techniques

We expanded our study with other techniques for measuring similarity including a range of libraries that measure textual similarity: difflib [Python Software Foundation, 2016] compares text sequences using Gestalt pattern matching, Python

NGram [Poulter, 2012] compares text sequences via fuzzy search using n-grams, FuzzyWuzzy [Cohen, 2011] uses fuzzy string token matching, jellyfish [Turk and Stephens, 2016] does approximate and phonetic matching of strings, and cosine similarity from scikit-learn [Pedregosa et al., 2011] which is a machine learning library providing data mining and data analysis, Java implementation of *n*-gram-based similarities using Jaccard index, Sorensen-Dice coefficient, and Cosine similarity [Debatty, 2018]. We also employed diff, the classic file comparison tool, and bsdiff, a binary file comparison tool. Using diff or bsdiff, we calculate the similarity between two Java files *x* and *y* using

$$\text{sim}_D(x, y) = 1 - \frac{\min(|y|, |D(x, y)|)}{|y|} \quad (5.5)$$

where $D(x, y)$ is the output of *diff* or *bsdiff*⁶ and $|D(x, y)|$ is the number of lines in the *diff* or *bsdiff* output.

The result of $\text{sim}_D(x, y)$ is asymmetric as it depends on the size of the denominator. Hence $\text{sim}_D(x, y)$ usually produces a different result from $\text{sim}_D(y, x)$. This is because $\text{sim}_D(x, y)$ provides the distance of editing *x* into *y* which is different in the opposite direction.

The summary of all selected tools and their respective similarity measurement methods are presented in Table 5.2. The default configurations of each tools, as displayed in Table 5.3, are extracted from (1) the values displayed in the help menu of the tools, (2) the tools' websites, (3) or the tools' papers (e.g., Deckard [Jiang et al., 2007b]). The range of parameter values we searched for in our study are also included in Table 5.3. In addition, we will write the name of each tool using only lower-case letters to show that the tool has been executed on the OCD and the SOCO data set . For example, the results of running NiCad tool will be denoted as “nicad” and JPlag will be denoted as “jplag”.

⁶We use the raw text output from both tools. The *diff* output was from using the parameters **-i** **-E** **-b** **-w** **-B** **-e**. The *bsdiff* output was used as-is without any parameter.

Table 5.2: Tools with their similarity measures

Tool/Technique	Similarity calculation
Clone Det.	
ccfx [Kamiya et al., 2002]	tokens and suffix tree matching
deckard [Jiang et al., 2007b]	characteristic vectors of AST optimised by LSH
iclones [Göde and Koschke, 2009]	tokens and generalised suffix tree
nicad [Roy and Cordy, 2008]	TXL and string comparison (LCS)
simian [Harris, 2003]	line-based string comparison
sourcerercc [Sajnani et al., 2016]	token index with filtering heuristics
vincent [Ragkhitwetsagul et al., 2018b]	source code image comparison
Plagiarism Det.	
jplag-java [Prechelt et al., 2002]	tokens, Karp Rabin matching, Greedy String Tiling
jplag-text [Prechelt et al., 2002]	tokens, Karp Rabin matching, Greedy String Tiling
plaggie [Ahtiainen et al., 2006]	N/A (not disclosed)
sherlock [Pike and Loki, 2002]	digital signatures
simjava [Gitchell and Tran, 1999]	tokens and string alignment
simtext [Gitchell and Tran, 1999]	tokens and string alignment
Compression	
7zncd	NCD with 7z
bzip2ncd	NCD with bzip2
gzipncd	NCD with gzip
xz-ncd	NCD with xz
icd	Equation 5.4
ncd [Cilibrasi et al., 2015]	ncd tool with bzlib & zlib
Others	
bsdiff	Equation 5.5
diff	Equation 5.5
difflib [Python Software Foundation, 2016]	Gestalt pattern matching
fuzzywuzzy [Cohen, 2011]	fuzzy string matching
jellyfish [Turk and Stephens, 2016]	approximate and phonetic matching of strings
ngram [Poulter, 2012]	fuzzy search based using n-gram
cosine [Pedregosa et al., 2011]	cosine similarity from machine learning library
jaccard [Debatty, 2018]	jaccard set similarity based on n-gram tokens
sorensen-dice [Debatty, 2018]	Sorenson-Dice set similarity based on n-gram tokens
ncosine [Debatty, 2018]	cosine similarity based on n-gram tokens

Table 5.3: Tools and parameters with chosen value ranges (DF denotes default parameters)

Tool	Settings	Details	DF	Range
Clone det.				
ccfx	b	min no. of tokens	50	3 4 5 10 15 16 17 18 19 20 21 22 23 24 25 30 35 40 45 50
deckard	t mintoken stride similarity	min token kinds min no. of tokens sliding window size clone similarity	12 50 inf 1.0	1 2 3 .. 14 30, 50 0, 1, 2, inf 0.90, 0.95, 1.00
iclones	minblock	min token length	20	8 10 20 30 40 50
nicad	minclone UPI minline rename abstract	min no. of tokens % of unique code min no. of lines variable renaming code abstraction	100 0.30 10 none none	50 60 .. 140 150 0.30, 0.50 5, 8, 10 blind, consistent none, declaration, statement, expression, condition, literal
simian	threshold options	min no. of lines other options	6 none	3 4 5 .. 10 none, ignoreCharacters, ignoreIdentifiers, ignoreLiterals, ignoreVariableNames
sourcererc	similarity	clone similarity	80	20, 40, 60, 80
Plagiarism det.				
jplag-java	t	min no. of tokens	9	1 2 3 .. 12
jplag-text	t	min no. of tokens	9	1 2 3 .. 12
plaggie	M	min no. of tokens	11	1 2 3 .. 14
sherlock	N Z	chain length zero bits	4 3	1 2 3 .. 8 0 1 2 .. 8
simjava	r	min run size	N/A	10 11 12 .. 24
simtext	r	min run size	N/A	4 5 6 .. 12
Compression				
7zncd-bzip2	mx	compression level	N/A	1 3 5 7 9
7zncd-deflate	mx	compression level	N/A	1 3 5 7 9
7zncd-deflate64	mx	compression level	N/A	1 3 5 7 9
7zncd-lzma	mx	compression level	N/A	1 3 5 7 9
7zncd-lzma2	mx	compression level	N/A	1 3 5 7 9
7zncd-ppmd	mx	compression level	N/A	1 3 5 7 9
bzip2ncd	C	block size	N/A	1 2 3 .. 9
gzipncd	C	compression speed	N/A	1 2 3 .. 9
icd	ma	compression algo.	N/A	bzip2, deflate, deflate64, lzma, lzma2, PPMD
ncd-zlib	mx	compression level	N/A	1 3 5 7 9
ncd-bzlib	N/A			
xzncd	N/A			
xzncd	-N	compression level	6	1 2 3 .. 9, e
Others				
bsdiff	N/A			
diff	N/A			
difflib	autojunk whitespace	auto. junk heuristic ignoring white space	N/A N/A	true, false true, false
fuzzywuzzy	similarity	similarity calculation	N/A	ratio, partial_ratio, token_sort_ratio, token_set_ratio
jellyfish	distance	edit distance algo.	N/A	jaro_distance, jaro_winkler
ngram	N/A			
cosine	N/A			
jaccard	N/A			
sorensen-dice	N/A			
ncosine	N/A			

Table 5.4: Descriptions of the 10 original Java classes in the generated OCD data set

No.	File	SLOC	Description
1	BubbleSort.java*	39	Bubble Sort implementation
2	EightQueens.java [†]	65	Solution to the Eight Queens problem
3	GuessWord.java*	115	A word guessing game
4	TowerOfHanoi.java*	141	The Tower of Hanoi game
5	InfixConverter.java*	95	Infix to postfix conversion
6	Kapreka_Transformation.java*	111	Kapreka Transformation of a number
7	MagicSquare.java [†]	121	Generating a Magic Square of size n
8	RailRoadCar.java*	71	Rearranging rail road cars
9	SLinkedList.java*	110	Singly linked list implementation
10	SqrtAlgorithm.java*	118	Calculating the square root of a number

* classes downloaded from <http://www.softwareandfinance.com/Java>

[†] classes downloaded from <http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures>

5.6 Experimental Scenarios

To answer the research questions, five experimental scenarios have been designed and studied following the framework presented in Figure 5.3. The experiment was conducted on a virtual machine with 2.67 GHz CPU (dual core) and 2 GB RAM running Scientific Linux release 6.6 (Carbon), and 24 Microsoft Azure virtual machines with up to 16 cores, 56 GB memory running Ubuntu 14.04 LTS. The details of each scenario are explained below.

5.6.1 Scenario 1 (Pervasive Modifications)

Scenario 1 studies tool performance against pervasive modifications (as simulated through source and bytecode obfuscation). At the same time, the best configuration for every tool is discovered. For this data set, we completed all the 5 steps of the OCD framework: data preparation, transformation, post-processing, similarity detection, and analysing the similarity report. However, post-processing is limited to pretty printing and no normalisation through decompilation is applied.

5.6.1.1 Preparation, Transformation, and Normalisation

This section follows Steps 1 and 2 in the framework. The original data consists of 10 Java classes: `BubbleSort`, `EightQueens`, `GuessWord`, `TowerOfHanoi`, `InfixConverter`, `Kapreka_Transformation`, `MagicSquare`, `RailRoadCar`, `SLinkedList`, and, finally, `SqrtAlgorithm`. We downloaded them from two

Table 5.5: Size of the data sets. The (generated) OCD data set in Scenario 1 has been compiled and decompiled before performing the detection in Scenario 2 ($\text{OCD}^{\text{decomp}}$). The SOCO data set is used in Scenario 3 and the SOCO with pervasive modification (SOCO^{ocd}) is used in Scenario 5.

Scenario	Data set	Files	#Comparisons	Positives	Negatives
1	OCD	100	10,000	1,000	9,000
2	$\text{OCD}^{\text{decomp}}$	100	10,000	1,000	9,000
3	SOCO	259	67,081	453	66,628
3	SOCO^{ocd}	330	20,691	1,045	19,646

programming websites as shown in Table 5.4 along with the class descriptions. We selected only the classes that can be compiled and decompiled without any required dependencies other than the Java SDK. All of them are short Java programs with less than 200 SLOC and they illustrate issues that are usually discussed in basic programming classes. The process of test data preparation and transformation is illustrated in Figure 5.6. First, we selected each original source code file and obfuscated it using Artifice. This produced the first type of obfuscation: source-level obfuscation (No. 1). An example of a method before and after source-level obfuscation by Artifice is displayed on the top of Figure 5.5 (formatting has been adjusted due to space limits).

Next, both the original and the obfuscated versions were compiled to bytecode, producing two bytecode files. Then, both bytecode files were obfuscated once again by ProGuard, producing two more bytecode files.

All four bytecode files were then decompiled by either Krakatau or Procyon giving back eight additional obfuscated source code files. For example, No. 1 in Figure 5.6 is a pervasively modified version via source code obfuscation with Artifice. No. 2 is a version which is obfuscated by Artifice, compiled, obfuscated with ProGuard, and then decompiled with Krakatau. No. 3 is a version obfuscated by Artifice, compiled and then decompiled with Procyon. Using this method, we obtained 9 pervasively modified versions for each original source file, resulting in 100 files for the data set. The only post-processing step in this scenario is normalisation through pretty printing. We call the generated data set *the OCD data set*.

```

/* original */
public MagicSquare(int n) {
    square=new int[n][n];
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++) {
            square[i][j]=0;
            ...
        }
}

/* original + Krakatau */
public MagicSquare(int i){
    super();
    this.square=new int[i][i];
    int i0=0;
    int i1=0;
    while(i1<i) {
        this.square[i0][i1]=0;
        i1=i1+1;
    }
    i0=i0+1;
    ...
}

/* original + Procyon */
public MagicSquare
    (final int n) {
    super();
    this.square = new int[n][n];
    for (int i=0;i<n;++i) {
        for (int j=0;j<n;++j) {
            {
                this.square[i][j]=0;
            }
        }
    }
    ...
}

/* ARTIFICE */
public MagicSquare(int v2) {
    f00=new int[v2][v2];
    int v3;
    v3=0;
    while(v3<v2) {
        int v4;
        v4=0;
        while(v4<v2) {
            f00[v3][v4]=0;
            v4=v4+1;
        }
        v3=v3+1;
        ...
    }
}

/* ARTIFICE + Krakatau */
public MagicSquare(int i){
    super();
    this.f00=new int[i][i];
    int i0=0;
    int i1=0;
    while(i1<i){
        this.f00[i0][i1]=0;
        i1=i1+1;
    }
    i0=i0+1;
    ...
}

/* ARTIFICE + Procyon */
public MagicSquare {
    (final int n) {
        super();
        this.f00=new int[n][n];
        for (int i=0;i<n;++i) {
            for (int j=0;j<n;++j)
                this.f00[i][j]=0;
        }
    }
    ...
}

```

Figure 5.5: The same code fragments, a constructor of `MagicSquare`, after pervasive modifications, and compilation/decompilation.

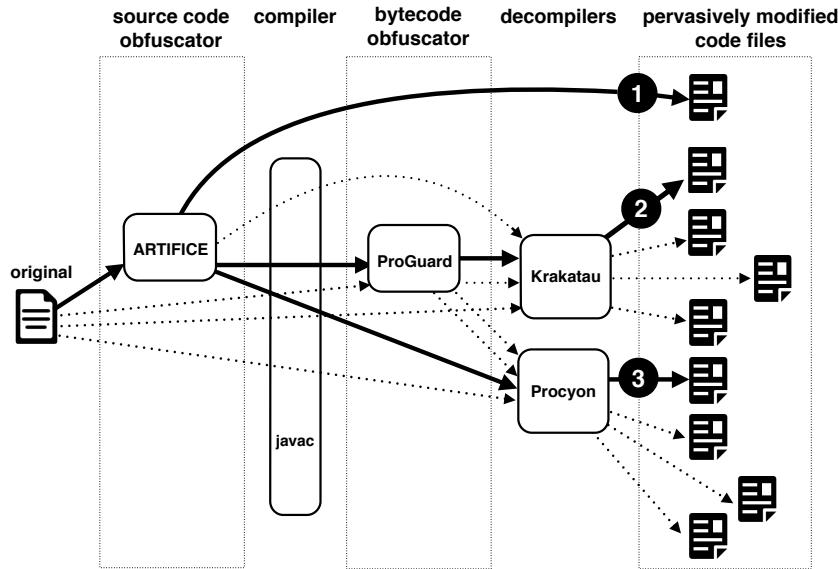


Figure 5.6: Test data generation process

5.6.1.2 Similarity Detection

The generated OCD data set of 100 Java code files is used for pairwise similarity detection in Step 4 of the framework in Figure 5.3, resulting in 10,000 pairs of source code files with their respective similarity values. We denote each pair and their similarity as a triple (x, y, sim) . Since each tool can have multiple parameters to adjust and we aimed to cover as many parameter settings as possible, we repeatedly ran each tool several times with different settings in the range listed in Table 5.3. Hence, the number of reports generated by one tool equals the number of combinations of its parameter values. A tool with two parameters $p_1 \in P_1$ and $p_2 \in P_2$ has $|P_1| \times |P_2|$ different settings. For example, sherlock has two parameters $N \in \{1, 2, 3, \dots, 8\}$ and $Z \in \{0, 1, 2, 3, \dots, 8\}$. We needed to do $8 \times 9 \times 10,000 = 720,000$ pairwise comparisons and generated 72 similarity reports. To cover the 34 tools with all of their possible configurations, we performed 14,950,000 pairwise comparisons in total and analysed 1,495 reports.

5.6.1.3 Analysing the Similarity Reports

In Step 5 of the framework, the results of the pairwise similarity detection are analysed. The 10,000 pairwise comparisons result in 10,000 (x, y, sim) entries. As in Equation 5.1, all pairs x, y are considered to be similar when the reported similarity

sim is larger than a threshold T . Such a threshold must be set in an informed way to produce sensible results. However, as the results of our experiment will be extremely sensitive to the chosen threshold, we want to use the optimal threshold, i.e., the threshold that produces the best results. Therefore, we vary the cut-off threshold T between 0 and 100.

As shown in Table 5.5, the ground truth of the generated data set contains 1,000 positives and 9,000 negatives. The positive pairs are the pairs of files generated from the same original code⁷. For example, all pairs that are the derivatives of `InfixConverter.java` must be reported as similar. The other 9,000 pairs are negatives since they come from different original source code files and must be classified as dissimilar. Using this ground truth, we can count the number of true and false positives in the results reported for each of the tools. We choose the F-score as the method to measure the tools' performance. The F-score is preferred in this context since the sets of similar files and dissimilar files are unbalanced and the F-score does not take true negatives into account⁸.

The F-score is the harmonic mean of precision (ratio of correctly identified reused pairs to retrieved pairs) and recall (ratio of correctly identified pairs to all the identified pairs):

$$\text{precision} = \frac{TP}{TP + FP} \quad \text{recall} = \frac{TP}{TP + FN}$$

$$\text{F-score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Using the F-score, we can search for the best threshold T under which each tool has its optimal performance with the highest F-score. For example in Figure 5.7, after varying the threshold from 0 to 100, `ncd-bzlib` has the best threshold $T = 37$ with the highest F-score of 0.846. Since each tool may have more than one parameter setting, we call the combination of the parameter settings and threshold that produces the highest F-score the tool's “optimal configuration”.

⁷In this study, we treat the files generated from the same original code as true positive because they share the same semantics. However, human may or may not consider them as similar since some of the code is heavily modified by obfuscation/compilation/decompilation.

⁸For the same reason, we decided against using Matthews correlation coefficient (MCC).

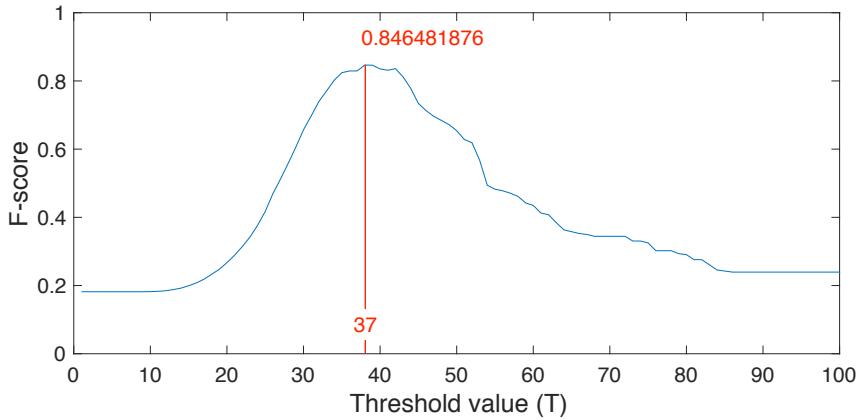


Figure 5.7: The graph shows the F-score and the threshold values of ncd-bzlib. The tool reaches the highest F-score when the threshold equals 37.

5.6.2 Scenario 2 (Reused Boiler-Plate Code)

In this scenario, we analyse the tools’ performance against an available data set that contains files in which fragments of boiler-plate code are reused with or without modifications. We choose the data set that has been provided by the Detection of SOurce COde Re-use competition for discovering monolingual re-used source code amongst a given set of programs [Flores et al., 2014], which we call the SOCO data set. We found that many of them share the same or very similar boiler-plate code fragments which perform the same task. Some of the boiler-plate fragments have been modified to adapt to the environment in which the fragments are re-used. Since we reused the data set from another study [Flores et al., 2014], we merely needed to format the source code files by removing comments and applying pretty-printing to them in Step 1 of our OCD framework (see Figure 5.3). We later skipped Step 2 and Step 3 of pervasive modifications and followed only Step 4 – similarity detection, and Step 5 – analysing similarity report in our framework.

We selected the Java training set containing 259 files for which the answer key of true clone pairs is provided. The answer key contains 84 file pairs that share boiler-plate code. Using the provided pairs, we are able to measure both false positives and negatives. For each tool, this data set produced $259 \times 259 = 67,081$ pairwise comparisons. Out of these 67,081 file pairs, $259 + 2 \times 84 = 427$ pairs are similar. However, after manually investigating false positives in a preliminary

study, we found that the provided ground truth contains errors. An investigation revealed that the provided answer key contained two large clusters in which pairs were missing and that two given pairs were wrong⁹. After removing the wrong pairs and adding the missing pairs, the corrected ground truth contains $259 + 2 \times 97 = 453$ pairs.

We performed two analyses on this data set: 1) applying the derived configurations to the data set and measuring the tools' performances, and 2) searching for the optimal configurations. Again, no transformation or normalisation has been applied to this data set as it is already prepared.

Since the SOCO data set is 2.59 times larger than the OCD data set (259 Java files vs. 100 Java files), it takes much longer to run. For example, it took CCFinderX 7 hours 48 minutes¹⁰ to complete $259^2 = 67,081$ pairwise comparisons with one of its configurations on our Azure virtual machine. To complete the search space of $20 \times 14 = 280$ CCFinderX's configurations, it took us 90 days. Executions of the 34 tools with all of their possible configurations cover 100,286,095 pairwise comparisons in total for this data set compared to 14,950,000 comparisons in Scenario 1. We analysed 1,495 similarity reports in total.

5.6.3 Scenario 3 (Decompilation)

We are interested in studying the effects of normalisation through compilation/decompilation before performing similarity detection. This is based on the observation that compilation has a normalising effect. Variable names disappear in bytecode and nominally different kinds of control structures can be replaced by the same bytecode, e.g., `for` and `while` loops are replaced by the same `if` and `goto` structures at the bytecode level.

Likewise, changes made by bytecode obfuscators may also be normalised by decompilers. Suppose a Java program P is obfuscated (transformed, T) into Q ($P \xrightarrow{T} Q$), then compiled (C) to bytecode B_Q , and decompiled (D) to source code Q' ($Q \xrightarrow{C} B_Q \xrightarrow{D} Q'$). This Q' should be different from both P and Q due to the

⁹The authors of the data set confirmed that the data set contains errors.

¹⁰User time measured by `/usr/bin/time -p` command.

changes caused by the compiler and decompiler. However, with the same original source code P , if it is compiled and decompiled using the same tools to create P' ($P \xrightarrow{C} B_P \xrightarrow{D} P'$), P' should have some similarity to Q' due to the analogous compiling/decompiling transformations made to both of them. Hence, one might apply similarity detection to find similarity $\text{sim}(P', Q')$ and get more accurate results than $\text{sim}(P, Q)$.

In this scenario, we focus on the OCD data set containing pervasive code modifications of 100 source code files generated in Scenario 1. However, we added normalisation through decompilation to the post-processing (Step 3 in the framework) by compiling all the transformed files using javac and decompiling them using either Krakatau or Procyon. We then followed the same similarity detection and analysis process in Steps 4 and 5. The results are then compared to the results obtained from Scenario 1 to observe the effects of normalisation through decompilation.

5.6.4 Scenario 4 (Ranked Results)

In our three previous scenarios, we compared the tools' performances using their optimal F-scores. The F-score offers a weighted harmonic mean of precision and recall. It is a set-based measure that does not consider any ordering of results. The optimal F-scores are obtained by varying the threshold T to find the highest F-score. We observed from the results of the previous scenarios that the thresholds are highly sensitive to each particular data set. Therefore, we had to repeat the process of finding the optimal threshold every time we changed to a new data set. This was burdensome but could be done since we knew the ground truth data of the data sets. The configuration problem for clone detection tools including setting thresholds has been mentioned by several studies as one of the threats to validity [Wang et al., 2001]. There has also been an initiative to avoid using thresholds at all for clone detection [Keivanloo et al., 2015]. Hence, we try to avoid the problem of threshold sensitivity affecting our results. Moreover, this approach also has applications in software engineering including finding candidates for plagiarism detection, automated software repair, working code examples, and large-scale code

clone detection.

Instead of looking at the results as a set and applying a cut-off threshold to obtain true and false positives, we consider only a subset of the results based on their rankings. We adopt three error measures mainly used in information retrieval: precision at n (prec@ n), average r -precision (ARP), and mean average precision (MAP) to measure the tools' performances. We present their definitions below.

Given n as a number of top n results ranked by similarity, precision at n [Manning et al., 2009] is defined as:

$$\text{prec}@n = \frac{\text{TP}}{n}$$

In the presence of ground truth, we can set the value of n to be the number of relevant results (i.e., true positives). With a known ground truth, precision at n when n equals to the number of true positives is called r -precision (RP) where r stands for “relevant” [Manning et al., 2009]. If a set of relevant files for each query $q \in Q$ is $R_q = \{rf_{q_1}, \dots, rf_{q_n}\}$, then the r -precisions for a query q is:

$$\text{RP}_q = \frac{\text{TP}_q}{|R_q|}$$

With presence of more than one query, an average r -precision (ARP) can be computed as the mean of all r -precision values [Beitzel et al., 2009]:

$$\text{ARP} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \text{RP}_q$$

Lastly, mean average precision (MAP) measures the quality of results across several recall levels where each relevant result is returned. It is calculated from multiple average precision at n values where n_{q_i} is the number of retrieved results after each relevant result $rf_{q_i} \in R_q$ of a query q is found. An average precision at n (aprec@ n) of a query q is calculated from:

$$\text{aprec}@n_q = \frac{1}{|R_q|} \sum_{i=1}^{|R_q|} \text{prec}@n_{q_i}$$

Mean average precision (MAP) is then derived from the mean of all aprec@n values of all the queries in Q [Manning et al., 2009]:

$$\text{MAP} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \text{aprec}@n_{q_i}$$

Precision at n, ARP, and MAP are used to measure how well the tools retrieve relevant results within top- n ranked items for a given query [Manning et al., 2009]. We simulate a querying process by 1) running the tools on our data sets and generating similarity pairs, and 2) ranking the results based on their similarities reported by the tools. The higher the similarity value, the higher the rank. The top ranked result has the highest similarity value. If a tie happens, we resort to a ranking by alphabetical order of the file names.

For precision at n, the query is “*what are the most similar files in this data set?*” and we inspect only the top n results. Our calculation of precision at n in this study can be considered as a hybrid between a set-based and a ranked-based measure. We put the results from different original files in the same “set” and we “rank” them by their similarities. This is suitable for a case of plagiarism detection. To locate plagiarised source code files, a human investigator may not want to give a specific file as a query (since he or she does not know which file has been copied) but he or she wants to retrieve a set of all similar pairs in a set ranked by their similarities. JPlag uses this method to report plagiarised source code pairs [Prechelt et al., 2002]. Moreover, finding the most similar files is useful in a manual study of large-scale code clones (e.g., in a study by Yang et al. [2017]) when too many clones are reported and researchers are only feasibly able to investigate by hand a few of the most similar clone candidates.

ARP and MAP are calculated by considering the question “*what are the most similar files for each given query q ?*” For example, since we had a total of 100 files in our OCD data set, we queried 100 times. We picked one file at a time from the data set as a query and retrieved a ranked result of 100 files (including the query itself) according to the query. An r -precision was calculated from the top 10 results. We limited results to only the top 10, since our ground truth contained 10

pervasively modified versions for each original source code file (including itself). Thus, the number of relevant results, r , is 10 in this study. We derive ARP from the average of the 100 r -precision values. The same process is repeated for MAP except using average precision at n instead of r -precision. The query-based approach is suitable when one does not require the retrieval of all the similar pairs of code but only the most relevant ones for a given query. This situation occurs when performing code search for automated software repair [Ke et al., 2015]. One may not feasibly try all returned repair candidates but only the top-ranked ones. Another example is searching for working code examples [Keivanloo et al., 2014] when one wants to pick only the top ranked solution.

Using these three error measures, we can compare performances of the similarity detection techniques and tools without relying on the threshold at all. They also provide another aspect of evaluating the tools' performances by observing how well the tools report correct results within the top n pairs.

5.6.5 Scenario 5 (Pervasive Modifications + Boiler-plate Code)

We have two objectives for this experimental scenario. First, we are interested in a situation where local and global code modifications are combined together. This is done by applying pervasive modifications (global) on top of reused boiler-plate code (local). This scenario occurs in software plagiarism when only a small fragment of code is copied and later pervasive modifications are applied to the whole source code to conceal the copied part of the code. It also represents a situation where a boiler-plate code has been reused and repeatedly modified (or refactored) during software evolution. We are interested to see if the tools can still locate the reused boiler-plate code. Second, we shift our focus from measuring how well our tools find *all* similar pairs of pervasively modified code pieces, as we did in Scenario 1, to measuring how well our tools find similar pairs of code pieces based on *each* pervasive code modification type. This is a finer-grained result and provides insights into the effects of each pervasive code modification type on code similarity. The default configurations are chosen for this experimental scenario to reflect a real use case when one does not know the optimal configurations of the tools and also to

show the effect of each pervasive code modifications on the tools' performances when they are picked off-the-shelf without any tuning. Since some threshold needs to be chosen, we used the optimal threshold for each tool.

We use the data set called SOCO^{ocd} which is derived from the SOCO data set used in Scenario 3. We follow the 5 steps in our OCD framework (see Figure 5.3) by using the SOCO's data set with boiler-plate code as a test data (Step 1). Among 259 SOCO files, 33 are successfully compiled and decompiled after code obfuscations by our framework. Each of the 33 files generates 10 pervasively modified files (including itself) resulting in 330 files available for detection (Step 4). The statistics of SOCO^{ocd} is shown in Table 5.5.

We change the similarity detection in Step 4 to focus only on comparing modified code to their original. Given M as a set of the 10 pervasive code modification types, a set of similar pairs of files $\text{Sim}_m(F)$ out of all files F with a pervasive code modification m is

$$\begin{aligned} M &= \{O, A, K, P_c, P_gK, P_gP_c, AK, AP_c, AP_gK, AP_gP_c\} \\ \text{Sim}_m(F) &= \{(x, y) \in F_O \times F_m : m \in M; \text{sim}(x, y) > T\} \end{aligned} \quad (5.6)$$

Table 5.6 presents the 10 pervasive code modification types; including the original (O), source code obfuscation by Artifice (A), decompilation by Krakatau (K), decompilation by Procyon (P_c), bytecode obfuscation by ProGuard and decompilation by Krakatau (P_gK), bytecode obfuscation by ProGuard and decompilation by Procyon (P_gP_c), and four other combinations (AK , AP_c , AP_gK , AP_gP_c); and ground truth for each of them. The number of code pairs and true positive pairs of A to AP_gP_c are twice larger than the Original (O) type because of asymmetric similarity between pairs, i.e., $\text{Sim}(x, y)$ and $\text{Sim}(y, x)$.

We measured the tools' performance on each $\text{Sim}_m(F)$ set. By applying tools on a pair of original and pervasively modified code, we measure the tools based on one particular type of code modifications at a time. In total, we made 703,494 pairwise comparisons and analysed 34 similarity reports in this scenario.

Table 5.6: 10 pervasive code modification types

Type	Modification	Source	Obfuscation Bytecode	Decomp.	Pairs	TP
O	Original				1,089	55
A	Artifice		✓		2,178	110
K	Krakatau			✓	2,178	110
P_c	Procyon			✓	2,178	110
P_gK	ProGuard + Krakatau		✓	✓	2,178	110
P_gP_c	ProGuard + Procyon		✓	✓	2,178	110
AK	Artifice + Krakatau	✓		✓	2,178	110
AP_c	Artifice + Procyon	✓		✓	2,178	110
AP_gK	Artifice + ProGuard + Krakatau	✓	✓	✓	2,178	110
AP_gP_c	Artifice + ProGuard + Procyon	✓	✓	✓	2,178	110

5.7 Results

We used the five experimental scenarios of pervasive modifications, decompilation, reused boiler-plate code, ranked results, and the combination of local and global code modification to answer the six research questions. The automatic execution of 34 similarity analysers using the OCD framework on the data sets along with searching for their optimal parameters took several months to complete. Then, we carefully observed and analysed the similarity reports and the results are discussed below in order of the six research questions.

5.7.1 RQ1: Performance Comparison

How well do current similarity detection techniques perform in the presence of pervasive source code modifications and boiler-plate code?

The results for this research question are collected from the experimental Scenario 1 (pervasive modifications) and Scenario 2 (reused boiler-plate code).

5.7.1.1 Pervasively Modified Code

A summary of the tools' performances and their optimal configurations on the OCD data set are listed in Table 5.7. We show seven error measures in the table including false positives (FP), false negatives (FN), accuracy (Acc), precision (Prec), recall (Rec), area under ROC curve (AUC), and F-score (F1). The tools are classified into 4 groups: clone detection tools, plagiarism detection tools, compression tools, and

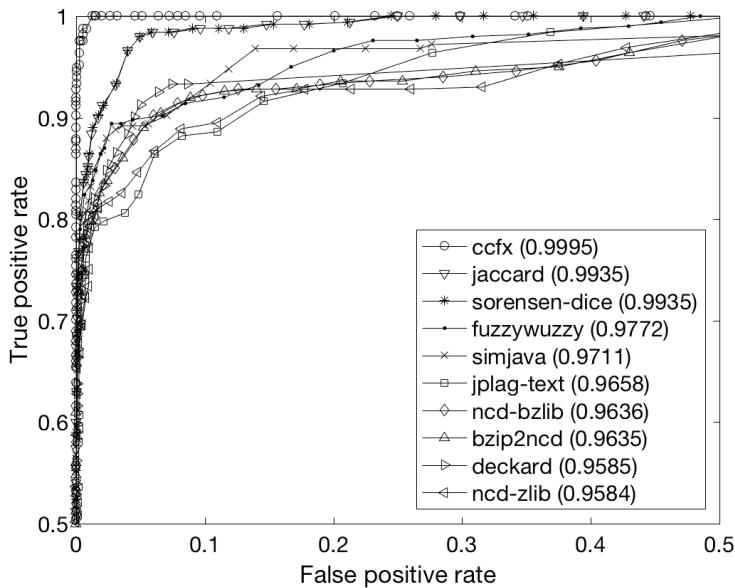


Figure 5.8: The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC).

other similarity analysers. We can see that the tools’ performances vary over the same data set. For clone detectors, we applied three different granularity levels of similarity calculation: line (L), token (T), and character (C). We find that measuring code similarity at different code granularity levels has an impact on the performance of the tools. For example, ccfx gives a higher F-score when measuring similarity at character level than at line or token level. We present only the results for the best granularity level in each case here. The complete results of the tools can be downloaded from the study website [Ragkhitwetsagul and Krinke, 2017], including the OCD data set before and after compilation/decompilation.

In terms of accuracy and F-score, the token-based clone detector ccfx is ranked first. The top 10 tools with highest F-score include ccfx (0.9760) followed by jaccard (0.8876), sorensen-dice (0.8873), fuzzywuzzy (0.876), jplag-java (0.8636), difflib (0.8629), simjava (0.8618), deckard (0.8509), bzip2ncd (0.8494), and ncd-bzlib (0.8465) respectively. Interestingly, tools from all the four groups appear in the top ten.

For clone detectors, we have a token-based tool (ccfx) and an AST-based tool (deckard) in the top ten. This shows that with pervasive modifications, multiple clone detectors with different detection techniques can offer comparable results

Table 5.7: OCD data set (Scenario 1): rankings (R**) by F-scores (**F1**) and optimal configuration of every tool and technique.**

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
Clone det.										
ccfx (C)*	b=5,t=11	36	24	24	0.9952	0.9760	0.9760	0.9995	0.9760	1
deckard (T)*	mintoken=30 stride=2 similarity=0.95	17	44	227	0.9729	0.9461	0.7730	0.9585	0.8509	8
iclones (L)*	minblock=10 minclone=50	0	36	358	0.9196	0.9048	0.4886	0.7088	0.6345	30
nicad (L)*	UPI=0.50 minline=8 rename=blind abstract=literal	38	38	346	0.9616	0.9451	0.6540	0.8164	0.7730	26
simian (C)*	threshold=4 ignoreVariableNames	5	150	165	0.9685	0.8477	0.8350	0.9262	0.8413	11
sourcererc (T)*	similarity=40	21	232	205	0.9563	0.7741	0.7950	0.9337	0.7844	25
Plag. det.										
jplag-java	t=7	19	58	196	0.9746	0.9327	0.8040	0.9563	0.8636	5
jplag-text	t=4	14	66	239	0.9695	0.9202	0.7610	0.9658	0.8331	14
plaggie	M=8	19	83	234	0.9683	0.9022	0.7660	0.9546	0.8286	17
sherlock	N=4, Z=2	6	142	196	0.9662	0.8499	0.8040	0.9447	0.8263	19
simjava	r=16	15	120	152	0.9728	0.8760	0.8480	0.9711	0.8618	7
simtext	r=4	14	38	422	0.9540	0.9383	0.5780	0.8075	0.7153	28
Compression										
7zncd-bzip2	mx=1,3,5	45	64	244	0.9692	0.9220	0.7560	0.9557	0.8308	16
7zncd-deflate	mx=7	38	122	215	0.9663	0.8655	0.7850	0.9454	0.8233	22
7zncd-deflate64	mx=7,9	38	123	215	0.9662	0.8645	0.7850	0.9453	0.8229	23
7zncd-lzma	mx=7,9	41	115	213	0.9672	0.8725	0.7870	0.9483	0.8275	18
7zncd-lzma2	mx=7,9	41	118	213	0.9669	0.8696	0.7870	0.9482	0.8262	20
7zncd-ppmd	mx=9	42	140	198	0.9662	0.8514	0.8020	0.9467	0.8260	21
bzip2ncd	C=1..9	38	62	216	0.9722	0.9267	0.7840	0.9635	0.8494	9
gzipncd	C=7	31	110	203	0.9687	0.8787	0.7970	0.9556	0.8359	13
icd	ma=lzma2 mx=7,9	50	86	356	0.9558	0.8822	0.6440	0.9265	0.7445	27
ncd-zlib	N/A	30	104	207	0.9689	0.8841	0.7930	0.9584	0.8361	12
ncd-bzlib	N/A	37	82	206	0.9712	0.9064	0.7940	0.9636	0.8465	10
xzncd	-e	39	120	203	0.9677	0.8691	0.7970	0.9516	0.8315	15
Others										
bsdiff*	N/A	71	199	577	0.9224	0.6801	0.4230	0.8562	0.5216	34
diff (C)*	N/A	8	626	184	0.9190	0.5659	0.8160	0.9364	0.6683	29
difflib	whitespace=false autojunk=false	28	12	232	0.9756	0.9846	0.7680	0.9412	0.8629	6
fuzzywuzzy	token_set_ratio	85	58	176	0.9766	0.9342	0.8240	0.9772	0.8757	4
jellyfish	jaro_distance	78	340	478	0.9182	0.6056	0.5220	0.8619	0.5607	33
ngram	N/A	49	110	224	0.9666	0.8758	0.7760	0.9410	0.8229	23
cosine	N/A	48	292	458	0.9250	0.6499	0.5420	0.9113	0.5911	32
jaccard	N/A	40	108	116	0.9776	0.8911	0.8840	0.9935	0.8876	2
sorensen-dice	N/A	57	116	110	0.9774	0.8847	0.8900	0.9935	0.8873	3
ncosine	N/A	65	784	226	0.8990	0.4968	0.7740	0.9317	0.6052	31

* — Tools that do not report similarity value directly. Similarity is measured at the granularity level of line (L), token (T), or character (C).

given their optimal configurations are provided. However, some clone detectors, e.g., iclones, nicad, and sourcererc did not perform well in this data set. ccfx performs the best – possibly due to a combination of using a suffix tree matching algorithm on a small number of tokens ($b=5$). This means that ccfx performs similarity computation on one small chunk of code at a time. This approach is flexible and effective in handling code with pervasive modifications that spread changes over the whole file. We also manually investigated the similarity reports of poorly performing iclones and nicad and found that the tools were susceptible to code changes involving the two decompilers, Krakatau and Procyon. When comparing files after decompilation by Krakatau to Procyon with or without bytecode obfuscation, they could not find any clones and hence reported zero similarity.

For plagiarism detection tools, jplag-java and simjava, which are token-based plagiarism detectors, are the leaders. Other plagiarism detectors give acceptable performance except simtext. This is expected since the tool is intended for plagiarism detection on natural text rather than source code. Compression tools show promising results using NCD for code similarity detection. They are ranked mostly in the middle from 9th to 27th with comparable results. The three bzip2-based NCD implementations, ncd-zlib, ncd-bzlib, and bzip2ncd only slightly outperform other compressors like gzip or lzma. So the actual compression method may not have a strong effect in this context. Other techniques for code similarity offer varied performance. Tools such as ngram, diff, cosine, ncosine, jellyfish and bsdiff perform badly. They are ranked among the last positions at 23th, 29th, 31st, 32nd, 33rd, and 34th respectively. Surprisingly, two Java tools using Jaccard and Sorensen-Dice coefficients on n -grams and two Python tools using difflib and fuzzywuzzy string matching techniques produce very high F-scores.

To find the overall performance over similarity thresholds from 0 to 100, we drew the receiver operating characteristic (ROC) curves, calculated the area under the curve (AUC), and compared them. The closer the value is to one, the better the tool’s performance. Figure 5.8 includes the ten highest AUC valued tools. We can

see from the figure that ccfx is again the best performing tool with the highest AUC (0.9995), followed by jaccard (0.9935) and sorensen-dice (0.9935), fuzzywuzzy (0.9772), simjava (0.9711), jplag-text (0.9658), ncd-bzlib (0.9636), and bzip2ncd (0.9635). The two other tools, deckard and ncd-zlib offer AUCs of 0.9585 and 0.9584.

The best tool with respect to accuracy, and F-score is ccfx. The tool with the lowest false positive is difflib. The lowest false negatives is given by diff. However, considering the large amount of false positive for diff (8,810 false positives which mean 8,810 out of 9,000 dissimilar files are treated as similar), the tool tends to judge everything as similar. The second lowest false negative is once again ccfx.

To sum up, we found that specialised tools such as source code clone and plagiarism detectors perform well against pervasively modified code. They were better than most of the compression-based and general string similarity tools. Compression-based tools mostly give decent and comparable results for all compression algorithms. String similarity tools perform poorly and mostly ranked among the last. However, we found that *n*-gram-based Jaccard and Sorensen-Dice and Python difflib and fuzzywuzzy perform surprisingly better than code clone detectors and plagiarsim detectors. They are both ranked highly among the top 5. Lastly, ccfx performed well on the data set, and is ranked the 1st on several error measures.

5.7.1.2 Boiler-plate Code

We report the complete evaluation of the tools on the SOCO data set with the optimal configurations in Table 5.8. Among the 34 tools, the top ranked tool in terms of F-score is jplag-text (0.9692), followed by simjava (0.9682), simian (0.9593) and jplag-java (0.9576). Most of the tools and techniques perform well on this data set. We observed high accuracy, precision, recall, and an F-score of over 0.7 for every tool except for diff and bsdiff. Since the data set contains source code that is copied and pasted with local modifications, the four clone detectors (ccfx, deckard, nicad, and simian) and three plagiarism detectors (jplag-text, jplag-java and simjava) performed very well with F-scores between 0.90 and 0.97. ccfx

Table 5.8: SOCO data set (Scenario 3): rankings (**R**) by F-scores (**F1**) and optimal configuration of every tool and technique.

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
Clone det.										
ccfx (C)*	b=15,16,17 t=12	25	42	15	0.9992	0.9125	0.9669	0.9905	0.9389	7
deckard (T)*	mintoken=50 stride=2 similarity=1.00	19	27	17	0.9993	0.9417	0.9625	0.9823	0.9520	5
iclones (L)*	minblock=40 minclone=50	19	20	57	0.9989	0.9519	0.8742	0.9469	0.9114	12
nicad (L)*	UPI=0.30 minline=5 rename=consistent abstract=condition	22	19	51	0.9990	0.9549	0.8874	0.9694	0.9199	9
simian (L)*	threshold=4 ignoreVariableNames	26	20	17	0.9994	0.9561	0.9625	0.9921	0.9593	3
sourcerercc (T)*	similarity=60	24	42	58	0.9985	0.9039	0.8720	0.9412	0.8876	14
Plag. det.										
jplag-java	t=12	29	26	13	0.9994	0.9442	0.9713	0.9895	0.9576	4
jplag-text	t=9	32	16	12	0.9996	0.9650	0.9735	0.9939	0.9692	1
plaggie	M=14	33	36	37	0.9989	0.9204	0.9183	0.9753	0.9193	10
sherlock	N=5, Z=0	22	22	54	0.9989	0.9477	0.8808	0.9996	0.9130	11
simjava	r=25	46	18	11	0.9996	0.9607	0.9757	0.9987	0.9682	2
simtext	r=12	17	73	19	0.9986	0.8560	0.9581	0.9887	0.9042	13
Compression										
7zncd-bzip2	mx=1,3,5	64	24	118	0.9979	0.9331	0.7395	0.9901	0.8251	30
7zncd-deflate	mx=7	64	27	97	0.9982	0.9295	0.7859	0.9937	0.8517	28
7zncd-deflate64	mx=7	64	27	96	0.9982	0.9297	0.7881	0.9957	0.8530	27
7zncd-lzma	mx=7,9	69	11	99	0.9984	0.9699	0.7815	0.9940	0.8655	24
7zncd-lzma2	mx=7,9	69	11	99	0.9984	0.9699	0.7815	0.9939	0.8655	24
7zncd-ppmd	mx=9	68	19	106	0.9981	0.9481	0.7660	0.9948	0.8474	29
bzip2ncd	C=1,2,3,..,8,9	54	20	94	0.9983	0.9473	0.7925	0.9944	0.8630	26
gzipncd	C=9	54	25	82	0.9984	0.9369	0.8190	0.9961	0.8740	19
icd [†]	ma=lzma mx=1,3	84	12	151	0.9976	0.9618	0.6667	0.9736	0.7875	31
ncd-zlib	N/A	57	10	91	0.9985	0.9731	0.7991	0.9983	0.8776	16
ncd-bzlib	N/A	52	30	82	0.9983	0.9252	0.8190	0.9943	0.8689	22
xzncd	2,3 6,7,8,9,e	64	13	94	0.9984	0.9651	0.7925	0.9942	0.8703	20
Others										
bsdiff	N/A	90	2125	212	0.9652	0.1019	0.5320	0.9161	0.1710	33
diff (C)	N/A	29	7745	5	0.8845	0.0547	0.9890	0.9180	0.1036	34
difflib	autojunk=true whitespace=true	42	30	21	0.9992	0.9351	0.9536	0.9999	0.9443	6
fuzzywuzzy	ratio	65	30	30	0.9991	0.9338	0.9338	0.9989	0.9338	8
jellyfish	jaro_distance	82	0	162	0.9976	1.0000	0.6424	0.9555	0.7823	32
ngram	N/A	59	20	84	0.9984	0.9486	0.8146	0.9967	0.8765	17
cosine	N/A	68	50	68	0.9982	0.8851	0.8499	0.9973	0.8671	23
jaccard	N/A	58	36	72	0.9984	0.9134	0.8411	0.9991	0.8759	18
sorense-dice	N/A	73	36	70	0.9984	0.9141	0.8455	0.9992	0.8784	15
ncosine	N/A	86	26	84	0.9984	0.9342	0.8146	0.9904	0.8703	21

* — Tools that do not report similarity value directly. Similarity is measured at the granularity level of line (L), token (T), or character (C).

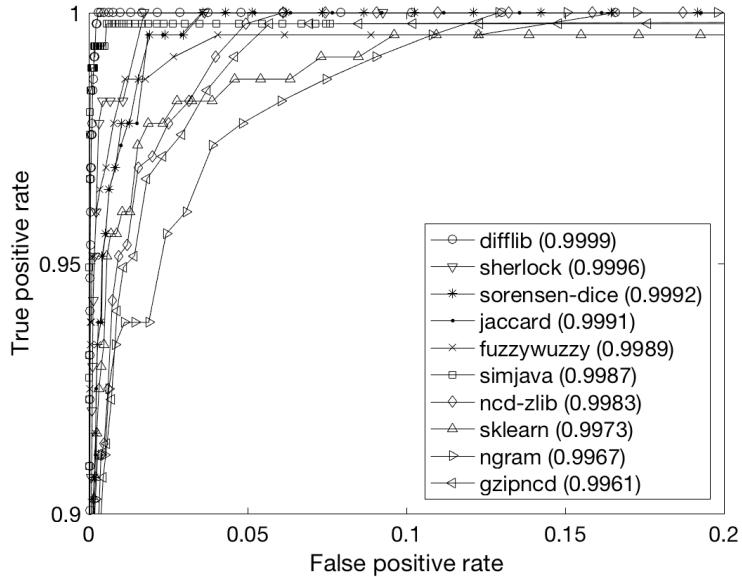


Figure 5.9: The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC) for SOCO.

and deckard produced the highest F-score when measuring similarity at character and token levels respectively. Other clone detectors including iclones, nicad, and simian provide the highest F-score at line level. The Python difflib and fuzzywuzzy are outliers of the Others group offering high performance against boiler-plate code with F-score of 0.9338 and 0.9443. Once again, these two string similarity techniques show promising results. The compression-based techniques are among the last although they still offer relatively high F-scores ranging from 0.8630 to 0.8776.

Regarding the overall performance over similarity thresholds of 0 to 100, the results are illustrated as ROC curves in Figure 5.9. The tool with the highest AUC is difflib (0.9999), followed by sherlock (0.9996), sorensen-dice (0.9992) and jaccard (0.9991).

To sum up, we observed that almost every tool detected boiler-plate code effectively by reporting high scores on all error measures. jplag-text, simjava, simian, jplag-java, and deckard are the top 5 tools for this data set in terms of F-score. Similar to pervasive modifications, we found the string matching techniques difflib and fuzzywuzzy ranked among the top 10.

5.7.1.3 Observations of the Tools’ Performances on the Two Data Sets

We can notice a clear distinction between the F-score rankings of clone/plagiarism detectors and string/compression-based tools on the SOCO data set. This is due to the nature of boiler-plate code that has local modifications, contained within a single method or code block on which clone and plagiarism detectors perform well. However, on a more challenging pervasive modifications data set, there is no clear distinction in terms of ranking between dedicated code similarity techniques, compression-based, and general text similarity tools. We found that the Java implementations of Jaccard and Sorensen-Dice n -gram similarity, Python difflib string matching, and Python fuzzywuzzy token similarity techniques even outperform several clone and plagiarism detection tools on both data sets. Provided that they are simple and easy-to-use Java and Python libraries, one can adopt these two techniques to measure code similarity in a situation where dedicated tools are not available (e.g., unparsable, incomplete methods or code blocks). Compression-based techniques are not ranked at the top in either scenario, possibly due to the small size of the source code – NCD is known to perform better with large files.

5.7.2 RQ2: Optimal Configurations

What are the best parameter settings and similarity thresholds for the techniques?

In the experimental Scenarios 1 and 2, we thoroughly analysed various configurations of every tool and found that some specific settings are sensitive to pervasively modified and boiler-plate code while others are not.

5.7.2.1 Pervasively Modified Code

The complete list of the best configurations of every tool for pervasive modifications from Scenario 1 can be found in the second column of Table 5.7. The optimal configurations are significantly different from the default configurations, in particular for the clone detectors. For example, using the default settings for ccfx ($b=50, t=12$) leads to a very low F-score of 0.5781 due to a very high number of false negatives.

Table 5.9: ccfx’s parameter settings for the highest precision and recall

Error measure	Value	ccfx’s parameters	
		<i>b</i>	<i>t</i>
Precision	1.000	19	7 8 9
Recall	0.980	5	12

Interestingly, a previous study on agreement of clone detectors [Wang et al., 2013b] observed the same difference between default and optimal configurations.

In addition, we performed a detailed analysis of ccfx’s configurations. This is because ccfx is a widely-used tool in several clone research studies. Two parameter settings are chosen for ccfx in this study: *b*, the minimum length of clone fragments in the unit of tokens, and *t*, the minimum number of kinds of tokens in clone fragments. We initially observed that the optimal F-scores of the tool were at either *b*=5 or *b*=19. Hence, we expanded the search space of ccfx parameters from 280 ($|b| = 20 \times |t| = 14$) to 392 settings ($|b| = 28 \times |t| = 14$) to reduce chances of finding a local optimum. We did a fine-grained search of *b* starting from 3 to 25 stepping by one and coarse-grained search from 30 to 50 stepping by 5.

From Figure 5.10, we can see that the default settings of ccfx, *b*=50 and *t*=12 (denoted with a \times symbol) provide a decent precision but very low recall. While there is no setting for ccfx to obtain the optimal precision and recall at the same time, there are a few cases in which ccfx can obtain high precision and recall as shown on the top right corner of Figure 5.10. Our derived ccfx’s optimal configuration is one of them. The best settings for precision and recall of ccfx are described in Table 5.9. The ccfx tool gives the best precision with *b*=19 and *t*=7, 8, 9 and gives the best recall with *b*=5 and *t*=12.

The landscape of ccfx performance in terms of F-score is depicted in Figure 5.11. Visually, we can distinguish regions that are the sweet spot for ccfx’s parameter settings against pervasive modifications from the rest. There are two regions covering the *b* value of 19 with *t* value from 7 to 9, and *b* value of 5 with *t* value from 11 to 12. The two regions provide F-scores ranging from 0.9589 up to 0.9760.

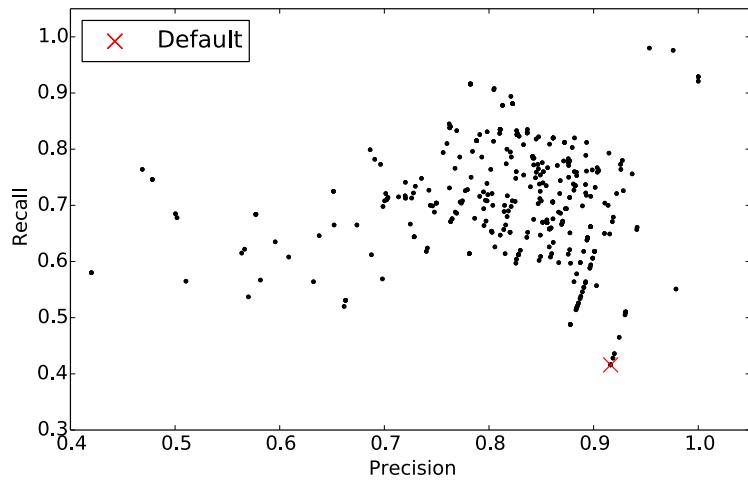


Figure 5.10: Trade off between precision and recall for 392 ccfx parameter settings. The default settings provide high precision but low recall against pervasive code modifications.

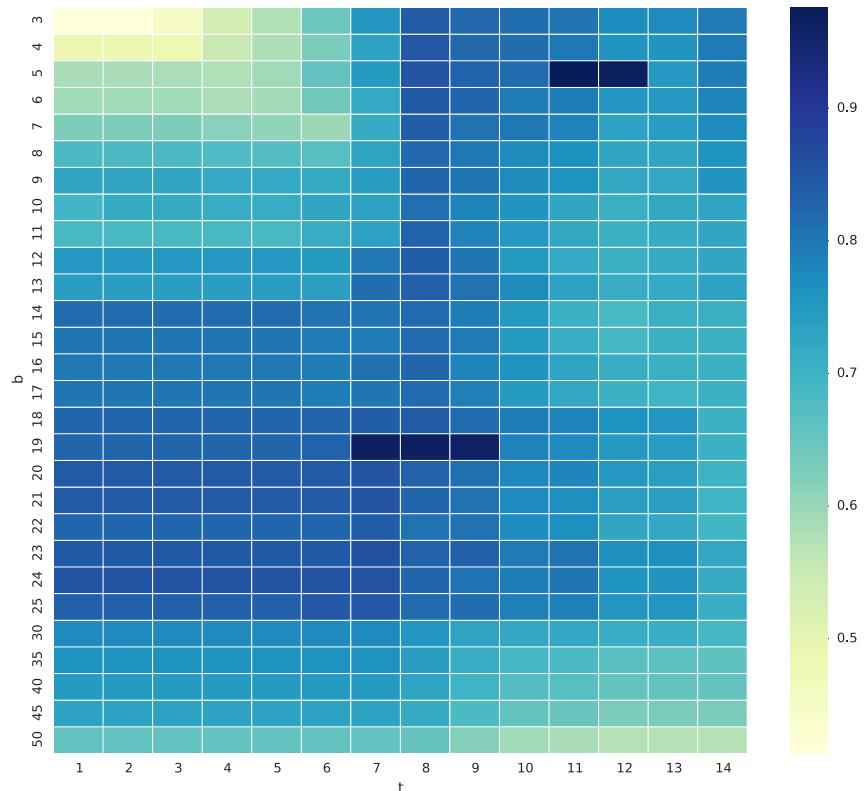


Figure 5.11: F-scores of 392 ccfx's b and t parameter values on pervasive code modifications

5.7.2.2 Boiler-plate Code

For boiler-plate code, we found another set of optimal configurations for the 34 tools by once again analysing a large search space of their configurations. The complete list of the best configurations for every tool from Scenario 3 can be found in the second column of Table 5.8. Similar to the OCD data set, the derived optimal configurations for SOCO are different from the tools' default configurations. For example, ccfx's best configurations have a smaller b , minimum number of tokens, of 15 compared to the default value of 50 while jplag-java's best configurations have a higher t value, the minimum number of tokens, of 12 compared to the default value of 9.

The results for both pervasively modified code and boiler-plate code show that the default configurations cannot offer the tools' their best performance. These empirical results support the findings of Wang et al. [2013b] that one cannot rely on the tools' default configurations. We suggest researchers and practitioners try their best to tune the tools before performing any benchmarking or comparisons of the tools' results to mitigate the threats to internal validity in their studies. Our optimal configurations can be used as guidelines for studies involving pervasive modifications and boiler-plate code. Nevertheless, they are only effective against their respective data set and not guaranteed to work well on other data sets.

5.7.3 RQ3: Normalisation by Decompilation

How much does compilation followed by decompilation as a pre-processing normalisation method improve detection results of pervasively modified code?

The results after adding compilation and decompilation for normalisation to the post-processing step before performing similarity detection on the generated data set in the experimental scenario 3 is shown in Figure 5.12. We can clearly observe that decompilation by both Krakatau and Procyon boosts the F-scores of every tool in the study.

Table 5.10 shows the performances of the tools after decompilation by Krakatau in terms of false positive (FP) rate, false negative (FN) rate, accuracy

Table 5.10: Optimal configuration of every tool obtained from the generated^{decomp} data set decompiled by Krakatau in Scenario 2 and their rankings (**R**) by F-scores (**F1**).

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
Clone det.										
ccfx*† (T)	b=5, t=8	50	0	18	0.9982	1.0000	0.9820	0.9991	0.9909	4
deckard*† (L)	mintoken=30 stride=1 similarity=0.95	29	0	84	0.9916	1.0000	0.9160	0.9459	0.9562	15
iclones* (L)	minblock=8 minclone=50	10	0	86	0.9914	1.0000	0.9140	0.9610	0.9551	17
nicad*† (T)	UPI=0.30 minline=8 rename=blind abstract=literal	19	0	106	0.9894	1.0000	0.8940	0.9526	0.9440	27
simian*† (T)	threshold=3 ignoreidentifiers	17	0	0	1.0000	1.0000	1.0000	0.9960	1.0000	1
sourcerercc* (T)	similarity=60	16	24	156	0.9820	0.9724	0.8440	0.9536	0.9036	32
Plagiarism det.										
jplag-java	t=4..12,default	23	0	0	1.0000	1.0000	1.0000	0.9964	1.0000	1
jplag-text	t=1	56	16	24	0.9960	0.9839	0.9760	0.9993	0.9799	8
plaggie	M=9	29	0	84	0.9916	1.0000	0.9160	0.9454	0.9562	16
sherlock	N=1,Z=0	60	34	22	0.9944	0.9664	0.9780	0.9989	0.9722	9
simjava†	r=18	17	0	0	1.0000	1.0000	1.0000	0.9998	1.0000	1
simtext	r=4; r=5	33	33	60	0.9907	0.9661	0.9400	0.9862	0.9529	19
Compression										
7zncd-bzip2	mx=1,3,5	49	40	40	0.9920	0.9600	0.9600	0.9983	0.9600	13
7zncd-deflate	mx=9	46	28	71	0.9901	0.9707	0.9290	0.9978	0.9494	21
7zncd-deflate64	mx=9	46	28	72	0.9900	0.9707	0.9280	0.9978	0.9489	22
7zncd-lzma	mx=7,9	48	28	72	0.9900	0.9707	0.9280	0.9977	0.9489	22
7zncd-lzma2	mx=7,9	48	28	72	0.9900	0.9707	0.9280	0.9977	0.9489	22
7zncd-ppmd	mx=9	49	40	31	0.9929	0.9604	0.9690	0.9985	0.9647	11
bzip2ncd	C=1..9,default	43	40	36	0.9924	0.9602	0.9640	0.9983	0.9621	12
gzipncd	C=8,9	38	28	63	0.9909	0.9710	0.9370	0.9980	0.9537	18
icd†	ma=lzma, mx=7,9	54	45	68	0.9887	0.9539	0.9320	0.9921	0.9428	28
ncd-zlib	N/A	37	28	72	0.9900	0.9707	0.9280	0.9981	0.9489	22
ncd-bzlib	N/A	42	46	36	0.9918	0.9545	0.9640	0.9984	0.9592	14
xzncd	-1	43	16	83	0.9901	0.9829	0.9170	0.9967	0.9488	26
Others										
bsdiff	N/A	78	0	171	0.9829	1.0000	0.8290	0.9595	0.9065	31
diff (C)	N/A	23	12	186	0.9802	0.9855	0.8140	0.9768	0.8916	33
difflib	autojunk=true	23	28	66	0.9906	0.9709	0.9340	0.9823	0.9521	20
fuzzywuzzy	token_set_ratio	90	0	32	0.9968	1.0000	0.9680	0.9966	0.9837	5
jellyfish	jaro_winkler	89	40	220	0.9740	0.9512	0.7800	0.9473	0.8571	34
ngram	N/A	60	48	104	0.9848	0.9492	0.8960	0.9726	0.9218	29
cosine	N/A	68	98	66	0.9836	0.9050	0.9340	0.9955	0.9193	30
jaccard	N/A	47	34	0	0.9966	0.9671	1.0000	0.9999	0.9833	6
sorensen-dice	N/A	64	34	0	0.9966	0.9671	1.0000	0.9999	0.9833	7
ncosine	N/A	80	50	8	0.9942	0.9520	0.9920	0.9990	0.9716	10

* — Tools that do not report similarity value directly. The similarity is measured at the granularity level of line (L), token (T), or character (C).

† — Tools that have several optimal configurations. The complete lists can be found in Appendix C and the study's website [Ragkhitwetsagul and Krinke, 2017].

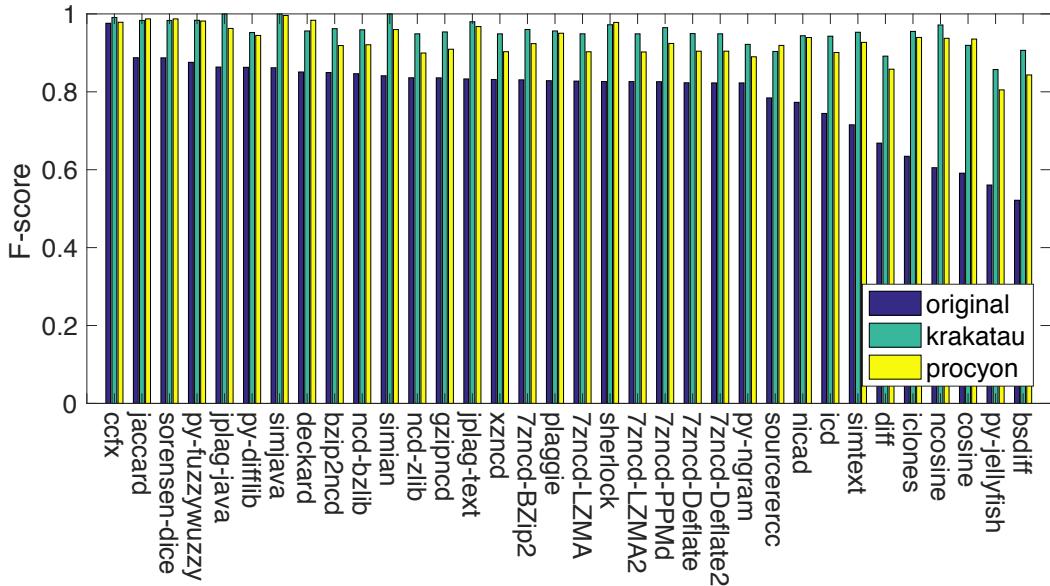


Figure 5.12: Comparison of tool performances (F1-score) before and after decompilation

Table 5.11: Wilcoxon signed-rank test of tools’ performances before and after decompilation by Krakatau and Procyon ($\alpha = 0.05$).

Test	p-value	Significant?	Effect size (A_{12})
Before-after decompiled by Krakatau	1.164e-10	Yes	0.972 (large)
Before-after decompiled by Procyon	1.164e-10	Yes	0.944 (large)

(Acc), precision (Prec), recall (Rec), area under ROC curve (AUC), and F-score. We can see that normalisation by compilation/decompilation has a strong effect on the number of false results reported by the tools. Every tool has its number of false positives and negatives greatly reduced and three tools, simian, jplag-java, and simjava, even no longer report any false results. All compression or other techniques still report some false results.

To strengthen the findings, we performed a statistical test to see if the performances before and after normalisation via decompilation differ with statistical significance. We chose the non-parametric two-tailed Wilcoxon signed-rank test [Wilcoxon, 1945]¹¹ and performed the test with a confidence interval value of 95% (i.e., $\alpha \leq 0.05$). Table 5.11 shows that the observed F-scores before and after decompilation are different with statistical significance for both Krakatau and

¹¹However, we also tried using the randomisation (i.e., permutation) test [Fisher, 1935, Box et al., 1978] on the results and found identical test results in all cases

Procyon. We complemented the statistical test by employing a non-parametric effect size measure called Vargha and Delaney's A_{12} measure [Vargha and Delaney, 2000] to measure the level of differences between two populations. We choose Vargha and Delaney's A_{12} measure because it is robust with respect to the shape of the distributions being compared [Thomas et al., 2014]. Put it another way, it does not require the two populations under comparison to be normally distributed, which is the case in our results of the tools' F1 scores. According to the guideline by Vargha and Delaney [2000], the A_{12} value of 0.5 means there is no difference between the two populations. A_{12} value over or below 0.5 means the first population outperforms the second population, and vice versa. The guideline shows that 0.56 is interpreted as small, 0.64 as medium, and 0.71 as large. Using this scale, our F-score differences after decompilation by Krakatau ($A_{12} = 0.972$) and Procyon ($A_{12} = 0.944$) compared to the original are large. According to the interpretation of A_{12} in Vargha and Delaney [2000], with Krakatau's A_{12} of 0.972 we can compute the probability that a random X_1 score from the set of tools' performance after decompilation by Krakatau will be greater than a random X_2 score from the set of tools' performance before decompilation by $2A_{12} - 1 = 2 \times 0.972 - 1 = 0.944$. This A_{12} effect size confirms that the tools' performance after decompilation by Krakatau will be higher than the original 94.4% of the time. The similar finding also applies to Procyon (88.8%). The large effect sizes clearly supports the findings that compilation and decompilation is an effective normalisation technique against pervasive modifications.

To gain insight, we carefully investigated the source code after normalisation and found that decompiled files created by Krakatau are very similar despite the applied obfuscation. As depicted in Figure 5.5 in the middle, the two code fragments become very similar after compilation and decompilation by Krakatau. This is because Krakatau has been designed to be robust with respect to minor obfuscations and the transformations made by Artifice and ProGuard are not very complex. Code normalisation by Krakatau resulted in multiple optimal configurations found for some of the tools. We selected only one optimal configuration to include in

Table 5.10 and separately reported the complete list of optimal configurations in Table C.1 in Appendix C.

Normalisation via decompilation using Procyon also improves the performance of the similarity detectors, but not as much as Krakatau (see Table 5.12). Interestingly, Procyon performs slightly better for deckard, sherlock, and cosine. An example of code before and after decompilation by Procyon is shown in Figure 5.5 at the bottom.

The main difference between Krakatau and Procyon is that Procyon attempts to produce much more high-level source code while Krakatau's is nearer to the bytecode. It seems that the low-level approach of Krakatau has a stronger normalisation effect. Hence, compilation/decompilation may be used as an effective normalisation method that greatly improves similarity detection between Java source code.

5.7.4 RQ4: Reuse of Configurations

Can we reuse optimal configurations from one data set in another data set effectively?

We answer this research question using the results from RQ1 and RQ2 (experimental Scenario 1 and 2 respectively). For the 34 tools from RQ1, we applied the derived optimal configurations obtained from the OCD data set (denoted as C_{ocd}) to the SOCO data set. Table 5.13 shows that using these configurations has a detrimental impact on the similarity detection results for another data set, even for tools that have no parameters (e.g., jaccard, sorensen-dice, and ncd-zlib) and are only influenced by the chosen similarity threshold. We noticed that the low F-scores when C_{gen} are reused on SOCO come from high number of false positives possibly due to their relaxed configurations.

To confirm this, we refer for the best configurations (settings and threshold) for the SOCO data set discussed in RQ1 (see Table 5.8), the comparison of best configurations between the two data sets is shown in Table 5.13. The reported F-scores are very high for the dataset-based optimal configurations (denoted as C_{soco}), confirming that configurations are very sensitive to the data set on which the

Table 5.12: Optimal configuration of every tool obtained from the generated^{decomp} data set (decompiled by Procyon) in Scenario 2 and their rankings (**R**) by F-scores (**F1**).

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
Clone det.										
ccfx* (L)	b=20, t=1..7	11	4	38	0.9958	0.9959	0.962	0.9970	0.9786	6
deckard* (T)	mintoken=30 stride=1, inf similarity=1.00	10	0	32	0.9968	1.0000	0.9680	0.9978	0.9837	4
iclones* (C)	minblock=10 minclone=50	0	18	98	0.9884	0.9804	0.9020	0.9508	0.9396	13
nicad* (W)	UPI=0.30 minline=10 rename=blind abstract= condition,literal	11	16	100	0.9884	0.9825	0.9000	0.9536	0.9394	14
simian* (C)	threshold=3 ignoreIdentifiers	23	8	70	0.9922	0.9915	0.9300	0.9987	0.9598	10
sourcerercc* (T)	similarity=60	11	16	136	0.9848	0.9818	0.8640	0.9990	0.9191	21
Plagiarism det.										
jplag-java	t=8	22	0	72	0.9928	1.0000	0.9280	0.9887	0.9627	9
jplag-text	t=9	11	16	48	0.9936	0.9835	0.9520	0.9982	0.9675	8
plaggie	M=13,14	10	16	80	0.9904	0.9829	0.9200	0.9773	0.9504	11
sherlock	N=1, Z=0	55	28	16	0.9956	0.9723	0.9840	0.9997	0.9781	7
simjava	r=default	11	8	0	0.9992	0.9921	1.0000	0.9999	0.9960	1
simtext	r=4	15	42	100	0.9858	0.9554	0.9000	0.9686	0.9269	17
	r=default		0							
Compression										
7zncd-bzip2	mx=1,3,5	51	30	116	0.9854	0.9672	0.8840	0.9909	0.9237	19
7zncd-deflate	mx=9	49	25	154	0.9821	0.9713	0.8460	0.9827	0.9043	24
7zncd-deflate64	mx=9	49	25	154	0.9821	0.9713	0.8460	0.9827	0.9043	24
7zncd-lzma	mx=7,9	52	16	164	0.9820	0.9812	0.8360	0.9843	0.9028	27
7zncd-lzma2	mx=7,9	52	17	164	0.9819	0.9801	0.8360	0.9841	0.9023	28
7zncd-ppmd	mx=9	53	22	122	0.9856	0.9756	0.8780	0.9861	0.9242	18
bzip2ncd	C=1..9,default	47	12	140	0.9848	0.9862	0.8600	0.9922	0.9188	22
gzipncd	C=3	36	40	133	0.9827	0.9559	0.8670	0.9846	0.9093	23
icd	ma=lzma, mx=7,9 ma=lzma2, mx=7,9	54	37	150	0.9813	0.9583	0.8500	0.9721	0.9009	29
ncd-zlib	N/A	41	30	158	0.9812	0.9656	0.8420	0.9876	0.8996	30
ncd-bzlib	N/A	47	8	140	0.9852	0.9908	0.8600	0.9923	0.9208	20
xzncd	-e	49	35	148	0.9817	0.9605	0.8520	0.9860	0.9030	26
Others										
bsdiff	N/A	73	48	236	0.9716	0.9409	0.7640	0.9606	0.8433	33
diff (C)	N/A	23	6	244	0.9750	0.9921	0.7560	0.9826	0.8581	32
difflib	autojunk=true	26	12	94	0.9894	0.9869	0.9060	0.9788	0.9447	12
fuzzywuzzy	token_set_ratio	90	0	36	0.9964	1.0000	0.9640	0.9992	0.9817	5
jellyfish	jaro_winkler	87	84	270	0.9646	0.8968	0.7300	0.9218	0.8049	34
ngram	N/A	58	8	192	0.9800	0.9902	0.8080	0.9714	0.8899	31
cosine	N/A	69	54	74	0.9872	0.9449	0.9260	0.9897	0.9354	16
jaccard	N/A	47	26	0	0.9974	0.9747	1.0000	0.9999	0.9872	2
sorenson-dice	N/A	64	26	0	0.9974	0.9747	1.0000	0.9999	0.9872	2
ncosine	N/A	79	34	88	0.9878	0.9641	0.9120	0.9972	0.9373	15

* — Tools that do not report similarity value directly. The similarity is measured at the granularity level of line (L), token (T), or character (C).

Table 5.13: Results after applying the best configurations (C_{gen}) from Scenario 1 to the SOCO data set and the derived best configurations for the SOCO set (C_{soco}).

Tools	Settings	C_{gen}			C_{soco}		
		T	generated F-score	SOCO F-score	Settings	T	SOCO F-score
ccfx (C)	b=5,t=11	36	0.9760	0.8441	b={15 16 17}, t=12	25	0.9389
jaccard	N/A	40	0.8876	0.4203	N/A	58	0.8759
sorensen-dice	N/A	57	0.8873	0.4134	N/A	73	0.8784
fuzzywuzzy	token_set_ratio	85	0.8757	0.6012	ratio	65	0.9338
jplag-java	t=7	19	0.8636	0.3168	t=12	29	0.9576
difflib	autojunk=false whitespace=false	28	0.8629	0.2113	autojunk=true whitespace=true	42	0.9443
simjava	r=16	15	0.8618	0.5888	r=25	46	0.9682
deckard (T)	M=30 S ₁ =2 S ₂ =0.95	17	0.8509	0.3305	M=50 S ₁ =1 S ₂ =1.0	19	0.9520
bzip2ncd	C=1..9	38	0.8494	0.3661	C=1 .. 9	54	0.8630
ncd-bzlib	N/A	37	0.8465	0.3357	N/A	52	0.8689

Note: M=mintoken, S₁=stride, S₂=similarity

similarity detection is applied. We found the dataset-based optimal configurations, C_{soco} , to be very different from the configuration for the generated data set C_{ocd} . The table shows only the top 10 tools from the OCD data set, but the same findings apply for every tool in our study.

Lastly, we noticed that the best thresholds for the tools are very different between one data set and another and that the chosen similarity threshold tends to have the largest impact on the performance of similarity detection. This observation provides further motivation for a threshold-free comparison using precision at n.

5.7.5 RQ5: Ranked Results

Which tools perform best when only the top n results are retrieved?

In experimental scenario 4, we applied three error measures; precision at n (prec@n), average r-precision (ARP) and mean average precision (MAP); adopted from information retrieval to the generated and SOCO data set. The results are discussed below.

Table 5.14: Top-10 rankings of using prec@n, ARP, and MAP over the OCD data set with the tools' optimal configurations

Rank	Pair-based		Query-based	
	F-score	prec@n	ARP	MAP
1	(0.976) ccfx	(0.976) ccfx	(1.000) ccfx	(1.000) ccfx
2	(0.888) jaccard	(0.891) jaccard	(0.927) sorensen-dice	(0.967) jaccard
3	(0.887) sorensen-dice	(0.890) sorensen-dice	(0.926) jaccard	(0.966) sorensen-dice
4	(0.876) fuzzywuzzy	(0.860) simjava	(0.915) fuzzywuzzy	(0.949) fuzzywuzzy
5	(0.864) jplag-java	(0.858) fuzzywuzzy	(0.913) ncd-bzlib	(0.943) ncd-bzlib
6	(0.863) difflib	(0.842) simian	(0.912) 7znacd-bzip2	(0.942) bzip2ncd
7	(0.862) simjava	(0.836) deckard	(0.909) bzip2ncd	(0.938) 7znacd-bzip2
8	(0.851) deckard	(0.836) jplag-java	(0.900) 7znacd-ppmd	(0.937) gzipncd
9	(0.849) bzip2ncd	(0.832) bzip2ncd	(0.900) gzipncd	(0.935) ncd-zlib
10	(0.847) ncd-bzlib	(0.828) difflib	(0.898) ncd-zlib	(0.933) jplag-text

Table 5.15: Top-10 rankings of using prec@n, ARP, and MAP over the SOCO data set with the tools' optimal configurations

Rank	Pair-based		Query-based	
	F-score	prec@n	ARP	MAP
1	(0.969) jplag-text	(0.965) jplag-text	(0.998) jplag-java	(0.997) jplag-java
2	(0.968) simjava	(0.960) simjava	(0.998) difflib	(0.997) difflib
3	(0.959) simian	(0.956) simian	(0.989) ccfx	(0.993) jplag-text
4	(0.958) jplag-java	(0.947) deckard	(0.989) simjava	(0.988) simjava
5	(0.952) deckard	(0.943) jplag-java	(0.987) gzipncd	(0.987) gzipncd
6	(0.944) difflib	(0.938) difflib	(0.986) jplag-text	(0.987) ncd-zlib
7	(0.939) ccfx	(0.929) ccfx	(0.985) ncd-zlib	(0.986) sherlock
8	(0.934) fuzzywuzzy	(0.929) fuzzywuzzy	(0.984) 7znacd-deflate	(0.986) 7znacd-deflate64
9	(0.920) nicad	(0.914) plaggie	(0.984) 7znacd-deflate64	(0.986) 7znacd-deflate
10	(0.919) plaggie	(0.901) nicad	(0.983) 7znacd-lzma	(0.984) fuzzywuzzy

5.7.5.1 Precision at n

As discussed in Section 5.6.4, we used prec@n in a pair-based manner. For the OCD data set, we sorted the 10,000 pairs of documents by their similarity values from the highest to the lowest. Then, we evaluated the tools based on a set of top n elements. We varied the value of n from 100 to 1500. In Table 5.14, we only reported the n equals to 1,000 since it is the number of true positives in the data set. The ccfx tool is ranked 1st with the highest prec@n of 0.976 followed by jaccard (0.891), and sorensen-dice (0.890). In comparison with the rankings for F-scores, the ranking of the ten tools changed slightly, as simjava and simian perform better while jplag-java and difflib tool now performed worse. ncd-zlib is no longer in the top 10.

As illustrated in Figure 5.13, varying fifteen n values of prec@n from 100 to

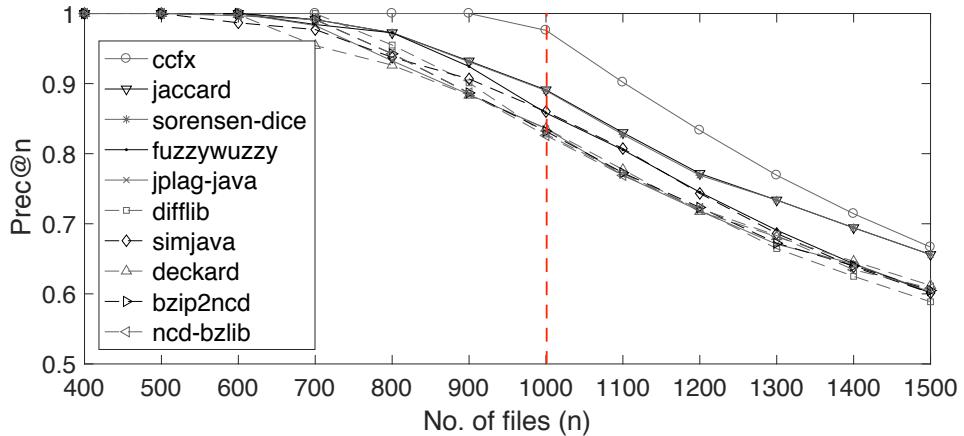


Figure 5.13: Precision-at-n of the tools according to varied numbers of n against the OCD data set

1500, stepping up by 100, gave us an overview of how well the tools perform across different n sizes. The number of true positives is depicted by a dotted line. We could see that most of the tools performed really well in the very first few hundreds of top n results by having steady flat lines at prec@n of 1.0 until the top 500 pairs. However, at the top 600 pairs, the performance of sorense dice, simjava, deckard, and ncd-bzlib started dropping. jaccard, py-fuzzywuzzy, jplag-java, and bzip2ncd started reporting false positives after the top 700 pairs while difflib could stay until the top 800 pairs. ccfx was the only tool that could maintain 100% correct results until the top 900 pairs. After that, it also started reporting false positives. At the top 1,500 pairs, all the tools offered prec@n at approximately 0.6 to 0.7. Due to a fairly small data set, this finding of perfect 1.0 prec@n until the first 500 pairs may not generalise to other data sets, as the similar performances achieved by the tools on the first 500 pairs might be due to intrinsic properties of the analysed programs.

For the SOCO data set, we varied the n value from 100 to 800, also stepping up by 100. The results in Table 5.15 used the n value of 453 which is the number of true positives in the corrected ground truth. We can clearly see that the ranking of 10 tools using prec@n closely resembles the one using F-scores. jplag-text is the top ranked tool followed by simjava, jplag-java, simian, and deckard. The ranking of eight tools is exactly the same as using F-score. jplag-java and nicad perform slightly worse using prec@453 and move down one position. The overall

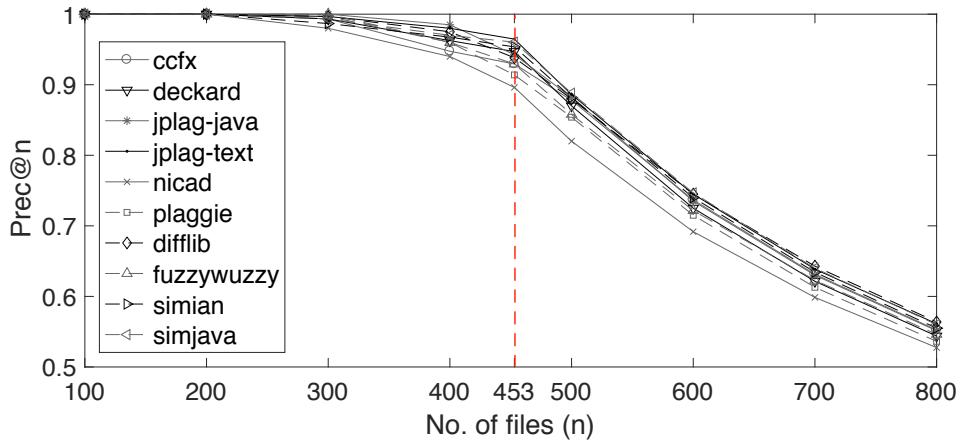


Figure 5.14: Precision-at-n of the tools according to varied numbers of n against SOCO data set

performances of the tools across various n values is depicted in Figure 5.14 with the dotted line representing the number of true positives. The chart is somewhat analogous to the OCD data set (Figure 5.13). Most of the tools started reporting false positives at the top 300 pairs except jplag-java and fuzzywuzzy. After the top 400 pairs, no tool could any longer maintain 100% true positive results.

Since $\text{prec}@n$ is calculated from a set of top- n ranked results, its value shows how fast a tool can retrieve correct answers to a limited set of n most similar files. It also reflects how well the tool can differentiate between similar and dissimilar documents. A good tool should not be confused and should produce a large gap in the similarity values between the true positive and the true negative results. In this study, ccfx and jplag-text have shown to be the best tools in terms of $\text{prec}@n$ for pervasive modifications and boiler-plate code respectively. They are also the best tools based on F-scores in RQ1.

5.7.5.2 Average r -Precision

ARP is a query-based error measure that needs knowledge of ground truth. Since we knew the ground truth for our two data sets, we did not need to vary the values of n as in $\text{prec}@n$. The value of n was set to the number of true positives.

For the OCD data set, each file in the set of 100 files was used as a query once. Each query received 100 files ranked by their similarity values. We knew the ground truth that each file has 10 other similar files including itself (i.e., r or the number

Table 5.16: One-tailed randomization test with 100K samples of the ARP values from the OCD data set.

Tool	ccfx	sorensen-dice	jaccard	fuzzywuzzy	ncd-bzlip	bzip2	bzip2ncd	ppmd	gzipncd	ncd-zlib
ccfx	►		►	►	►	►	►	►	►	►
sorensen-dice	□		□	□	□	□	□	□	□	□
jaccard	□	□		□	□	□	□	□	□	□
fuzzywuzzy	□	□	□		□	□	□	□	□	□
ncd-bzlib	□	□	□	□		□	□	□	□	□
bzip2	□	□	□	□	□		□	□	□	□
bzip2ncd	□	□	□	□	□	□		□	□	□
ppmd	□	□	□	□	□	□	□		□	□
gzipncd	□	□	□	□	□	□	□	□		□
ncd-zlib	□	□	□	□	□	□	□	□	□	

► — statistically significant difference of 1st tool's ARP (row) to 2ndnd tool's ARP (column), i.e., $\alpha \leq 0.05$.
 □ — no statistically significant difference.

of relevant documents equals 10). We cut off after the top 10 ranked results and calculated an r -precision value. Finally, we computed ARP from an average of the 100 r -precisions. We reported the ARPs of the ten tools in Table 5.14. We can see that ccfx is still ranked first with the perfect ARP of 1.000 followed by sorensen-dice, jaccard, and fuzzywuzzy. ncd-bzlib now performs much better using ARP and is ranked fifth. Interestingly, the 5th to 10th ranks are all compression-based tools. This shows that with the presence of pervasive modifications, code similarity using NCD-compression method is better at query-based results than most of the clone and plagiarism detectors and the string similarity tools.

For SOCO, only files with known, corrected, ground truth were used as queries. This is because ARP can only be computed when relevant answers are retrieved. We found that the 453 pairs in the ground truth were formed by 115 unique files, and we used them as our queries. The value of r here was not fixed as for the OCD data set. It depended on how many relevant answers existed in the ground truth for each particular query file and we calculated the r -precision based on that. The ARPs of the SOCO data set is reported in Table 5.15. jplag-java and difflib are ranked first with an ARP of 0.998, followed by ccfx and simjava both with an ARP of 0.989. Similar to the findings for the OCD data set, compression-based tools work well with a query-based approach by having 5 NCD tools ranked in the top 10.

Since ARP are computed based on means, we performed a statistical test to strengthen our results by testing for the statistical significance of differences in

Table 5.17: One-tailed randomization test with 100K samples of the MAP values from the OCD data set.

Tool	ccfx	jaccard	sorensen-dice	fuzzywuzzy	ncd-bzlib	bzip2ncd	bzip2	gzipncd	ncd-zlib	jplag-text
ccfx	►	►	►	►	►	►	►	►	►	►
jaccard	□	□	□	□	►	►	►	►	►	►
sorensen-dice	□	□	□	□	►	►	►	►	►	►
fuzzywuzzy	□	□	□	□	□	□	□	□	□	□
ncd-bzlib	□	□	□	□	□	□	□	□	□	□
bzip2ncd	□	□	□	□	□	□	□	□	□	□
bzip2	□	□	□	□	□	□	□	□	□	□
gzipncd	□	□	□	□	□	□	□	□	□	□
ncd-zlib	□	□	□	□	□	□	□	□	□	□
jplag-text	□	□	□	□	□	□	□	□	□	□

► — statistically significant difference of 1st tool’s MAP (row) to 2ndnd tool’s MAP (column), i.e., $\alpha \leq 0.05$.

□ — no statistically significant difference.

Table 5.18: One-tailed randomization test with 100K samples of the ARP values from the SOCO data set.

Tool	jplag-java	difflib	ccfx	simjava	gzipncd	jplag-text	ncd-zlib	deflate	deflate64	lzma
jplag-java	□	□	□	□	□	□	□	►	►	►
difflib	□	□	□	□	□	□	□	►	►	►
ccfx	□	□	□	□	□	□	□	□	□	□
simjava	□	□	□	□	□	□	□	□	□	□
gzipncd	□	□	□	□	□	□	□	□	□	□
jplag-text	□	□	□	□	□	□	□	□	□	□
ncd-zlib	□	□	□	□	□	□	□	□	□	□
deflate	□	□	□	□	□	□	□	□	□	□
deflate64	□	□	□	□	□	□	□	□	□	□
lzma	□	□	□	□	□	□	□	□	□	□

► — statistically significant difference of 1st tool’s ARP (row) to 2ndnd tool’s ARP (column), i.e. $\alpha \leq 0.05$.

□ — no statistically significant difference.

the set of r -precision values between tools. We chose a one-tailed non-parametric randomisation test (i.e., permutation test) due to its robustness in information retrieval as shown by Smucker et al. [2007]¹². We performed the test using 100,000 random samples with a confidence interval value of 95% (i.e., $\alpha \leq 0.05$). The statistical test results are shown in Table 5.16, Table 5.17, and Table 5.18. The tables are matrices of pairwise one-tailed statistical test results in the direction of rows \geq columns. The symbol ► represents statistical significance while the symbol □ represents no statistical significance. For example, in Table 5.16 and Table 5.17, the ► on the left most of the top row [ccfx, sorensen-dice] shows that the mean of r -precision values of ccfx are higher than or equal to sorensen-dice’s with statistical

¹²We also tried using one-tailed Wilcoxon signed-rank test on the results and found identical test results in all cases

significance. On the other hand, we can see that the mean of r -precision values of jaccard is higher than sorensen-dice with no statistical significance as represented by \square at the location of [jaccard, sorensen-dice].

For the OCD data set (Table 5.16), we found that ccfx is the only tool that dominates other tools on their r -precision values with statistical significance. For SOCO data set (Table 5.18), jplag-java and difflib outperform 7zncd-deflate, 7zncd-deflate64, and 7zncd-lzma with statistical significance.

ARP tells us how well the tools perform when we want all the true positive results in a query-based manner. For example, in automated software repair one wants to find similar source code given some original, buggy, source code that one possesses. One can use the original source code as a query and look for similar source files in a set of source code files. In our study, ccfx is the best tool for this retrieval method against pervasive modifications. jplag-java and difflib are the best tool for boiler-plate code.

5.7.5.3 Mean Average Precision

We included MAP in this study due to its well-known quality of discrimination and stability across several recall levels. It is also used when the ground truth for relevant documents is known. We computed MAP in a very similar way to ARP except that instead of only looking at the top r pairs, we calculated precision every time a new, relevant, source code file is retrieved. An average across all recall levels is then calculated. Lastly, the final average across all the queries is computed as MAP. We used the same number of relevant files as in the ARP calculations for the generated and the SOCO data set. The results for MAP are reported in Table 5.14 and Table 5.15.

For the OCD data set (Table 5.14), the rankings are very similar to those for ARP. ccfx, jaccard, and sorensen-dice are ranked 1st, 2nd and 3rd. For SOCO (Table 5.15), the rankings are very different to those obtained when using F-score and prec@n but similar to those for ARP. jplag-java and difflib become the best performers followed by jplag-text and simjava.

Compression-based tools are again found to offer good performance with

Table 5.19: One-tailed randomization test with 100K samples of the MAP values from SOCO data set.

Tool	jplag-java	difflib	jplag-text	simjava	gzipnacd	ncd-zlib	sherlock	deflate64	deflate	fuzzywuzzy
jplag-java	►	□	□	►	►	►	►	►	►	►
difflib	□	►	□	□	►	►	►	►	►	►
jplag-text	□	□	►	□	□	□	□	□	□	□
simjava	□	□	□	□	□	□	□	□	□	□
gzipnacd	□	□	□	□	□	□	□	□	□	□
ncd-zlib	□	□	□	□	□	□	□	□	□	□
sherlock	□	□	□	□	□	□	□	□	□	□
deflate64	□	□	□	□	□	□	□	□	□	□
deflate	□	□	□	□	□	□	□	□	□	□
fuzzywuzzy	□	□	□	□	□	□	□	□	□	□

► — statistically significant difference of 1st tool's MAP (row) to 2ndnd tool's MAP (column), i.e. $\alpha \leq 0.05$.
 □ — no statistically significant difference.

MAP. Five tools are ranked in the top 10 for both the generated and boiler-plate code data sets.

Similarly, since MAP is also computed based on mean, we performed a one-tailed non-parametric randomisation statistical test on pairwise comparisons of the tools' MAP values. The test results are shown in Table 5.17 and Table 5.19. For the OCD data set, we found the same results of ccfx dominating other tools' MAPs with statistical significance. Moreover, jaccard and sorensen-dice also statistically outperformed ncd-bzlib, bzip2ncd, 7zncd-bzip2, gzipnacd, ncd-zlib, and jplag-text. For the SOCO data set, we found that jplag-java and difflib outperform gzipnacd, ncd-zlib, sherlock, 7zncd-deflate64, 7zncd-deflate, and fuzzywuzzy with statistical significance.

MAP is similar to ARP because recall is taken into account. However, it differs from ARP by measuring precision at multiple recall levels. It is also different from F-score in terms of being query-based measure instead of a pair-based measure. It shows how well a tool performs on average when it has to find all true positives for each query. In this study, the best performing tool in terms of MAP is ccfx, followed by jaccard, for pervasively modified code and jplag-java and difflib for boiler-plate code respectively.

Table 5.20: F-scores of the tools on SOCO^{ocd} using the default configurations (with optimised threshold). **Highlighted** values have F-score higher than 0.8.

Tool	F-Score									
	O	A	K	P _c	P _g K	P _g P _c	A K	A P _c	A P _g K	A P _g P _c
Clone det.										
ccfx (C)*	0.8911	0.3714	0.0000	0.6265	0.0000	0.1034	0.0000	0.2985	0.0000	0.1034
deckard (T)*	0.9636	0.9217	0.1667	0.3333	0.0357	0.2286	0.1667	0.3252	0.0357	0.2286
iclones (L)*	0.5000	0.0000	0.0000	0.0357	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
nicad (T)*	0.5823	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
simian (L)*	0.8350	0.1034	0.0357	0.1356	0.0000	0.0357	0.0000	0.0357	0.0000	0.0357
sourcerercc (T)*	0.5679	0.0357	0.0000	0.0702	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Plagiarism det.										
jplag-java	1.0000	1.0000	0.7429	0.9524	0.2973	0.4533	0.7547	0.9720	0.2973	0.4507
jplag-text	0.9815	0.6265	0.5581	0.6304	0.3590	0.4250	0.4906	0.5581	0.3590	0.4304
plaggie	0.9636	0.9159	0.7363	0.9372	0.2171	0.4626	0.7363	0.9423	0.2171	0.4626
sherlock	0.9483	0.8298	0.7872	0.8298	0.3061	0.3516	0.6744	0.7826	0.3061	0.3516
simjava	0.9649	0.9815	1.0000	0.7525	0.3188	0.3913	0.8041	0.7525	0.3188	0.3913
simtext	0.9649	0.7191	0.1667	0.4932	0.0357	0.1667	0.0702	0.2258	0.0357	0.1667
Compression										
7zncd-bzip2	0.9273	0.7736	0.6852	0.8649	0.2446	0.3704	0.6423	0.7465	0.2446	0.3704
7zncd-deflate	0.9483	0.7579	0.6935	0.8406	0.2427	0.3333	0.6360	0.7418	0.2427	0.3333
7zncd-deflate64	0.9483	0.7579	0.6935	0.8406	0.2427	0.3333	0.6360	0.7373	0.2427	0.3333
7zncd-lzma	0.9649	0.7967	0.7488	0.8851	0.2663	0.3842	0.6768	0.7665	0.2632	0.3842
7zncd-lzma2	0.9649	0.7934	0.7536	0.8851	0.2718	0.3923	0.6700	0.7632	0.2697	0.4000
7zncd-ppmd	0.9623	0.7965	0.7628	0.8909	0.2581	0.3796	0.6667	0.8019	0.2581	0.3796
bzip2ncd	0.9649	0.8305	0.8302	0.9273	0.3590	0.4681	0.7612	0.8448	0.3562	0.4681
gzipncd	0.9623	0.7965	0.7628	0.8909	0.2581	0.3796	0.6667	0.8019	0.2581	0.3796
icd	0.9216	0.5058	0.4371	0.5623	0.2237	0.2822	0.3478	0.4239	0.2237	0.2822
ncd-zlib	0.9821	0.8571	0.8246	0.9432	0.4021	0.4920	0.7491	0.8559	0.3963	0.4920
ncd-bzlib	0.9649	0.8269	0.8269	0.9273	0.3529	0.4634	0.7500	0.8448	0.3500	0.4719
xzncd	0.9734	0.8416	0.7925	0.9198	0.3133	0.4615	0.7035	0.8148	0.3133	0.4615
Others										
bsdiff	0.4388	0.2280	0.1529	0.2005	0.1151	0.1350	0.1276	0.1596	0.1152	0.1353
diff (C)	0.2835	0.2374	0.1585	0.2000	0.1296	0.1248	0.1530	0.1786	0.1302	0.1249
difflib	0.9821	0.9550	0.8952	0.9565	0.4790	0.5087	0.8688	0.9381	0.4606	0.5091
fuzzywuzzy	1.0000	0.9821	0.9259	0.9636	0.4651	0.5116	0.9074	0.9541	0.4557	0.5116
jellyfish	0.9273	0.7253	0.6400	0.6667	0.2479	0.3579	0.5513	0.5000	0.2479	0.3662
ngram	1.0000	0.9464	0.8952	0.9346	0.4110	0.4490	0.8785	0.8908	0.4054	0.4578
cosine	0.9074	0.6847	0.7123	0.6800	0.3500	0.3596	0.5823	0.5287	0.3500	0.3596
jaccard	0.9636	0.8909	0.9273	0.9333	0.5000	0.5287	0.8727	0.8850	0.5000	0.5287
sorense-dice	0.9636	0.8909	0.9273	0.9333	0.5000	0.5287	0.8727	0.8772	0.5176	0.5287
ncosine	0.8750	0.6990	0.7629	0.7636	0.3846	0.4396	0.5825	0.6065	0.3846	0.4396

* — A tool that does not report similarity value directly. The similarity is measured at the granularity level of line (L), token (T), or character (C).

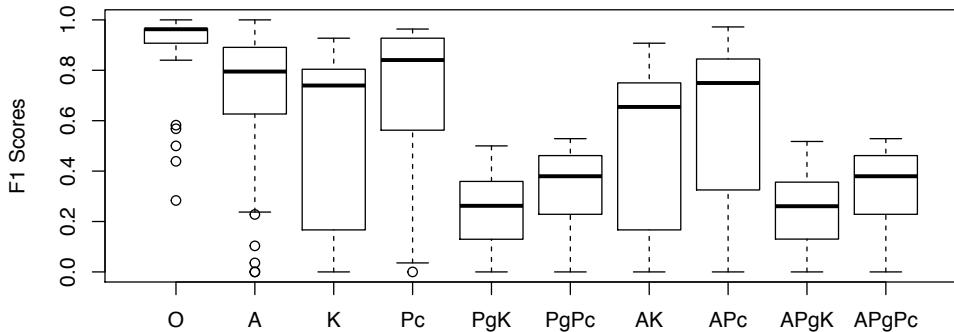


Figure 5.15: Distribution of tools performance for each pervasive modification type

5.7.6 RQ6: Pervasive Modifications + Boiler-plate Code

How well do the techniques perform when source code containing boiler-plate code has been pervasively modified?

Using the results from Experimental Scenario 5, we present the tools’ performances based on F-scores in Table 5.20 and show the distribution of F-scores in Figure 5.15. The F-scores are grouped according to the 10 pervasive code modification types (see Table 5.6). The numbers are highlighted when F-scores are higher than 0.8.

5.7.6.1 Tools’ Performances vs. Individual Pervasive Modification Type

On the original boiler-plate code without any modification (*O*), every tool except iclones, nicad, sourcerercc, bsdiff, and diff report high F-scores ranging from 0.8 to 1.0. This shows that most tools with their default configurations do not have a problem detecting boiler-plate code. The nicad tool performed poorly, possibly due to default configurations that aim at clones without variable renaming and code abstraction at all (i.e., set renaming=none and abstract=none). iclone’s default configurations of minimum 100 of clone tokens are too high compared to the optimal configurations of 40 found in RQ1. Similarly, sourcerercc’s default similarity threshold at 80% is probably too high for the data set compared to the optimal configuration at 60% found in RQ1. diff and bsdiff are too general to handle code with local modifications.

The tools perform worse after pervasive modifications are applied on top of

the boiler-plate code. Source code obfuscation by Artifice (A) has strong effects to ccfx, iclones, nicad, simian, sourcererc, bsdiff, and diff according to low F-scores of 0.0 to 0.2. deckard, jplag-java, plaggie, simjava, difflib, fuzzywuzzy and ngram maintained their high F-scores of over 0.9. Interestingly, jplag-java reported a perfect F-score of 1.0 possibly due to it being designed for detecting plagiarised code which is usually pervasively modified at source code level.

According to the boxplot in Figure 5.15, code after decompilation by Krakatau (K) results in lower F-scores than after decompilation by Procyon. Since the Krakatau decompilation process generates source code that is close to Java bytecode and mostly structurally different from the original, its generated code is challenging for tools that are based on lexical and syntactic similarity. In the group of clone detectors, ccfx, iclones, nicad, and sourcererc did not report any correct results at all (F-score = 0.0) while deckard and simian reported very low F-scores of 0.1667 and 0.0357 respectively. Code after decompilation by Procyon (P_c) had milder effects than Krakatau and Artifice. The tool simjava is the best for K with F-score of 1.000 and fuzzywuzzy is the best for P_c with F-score of 0.9636.

A combination of ProGuard and either Krakatau or Procyon (P_gK , P_gP_c) reported the lowest F-scores as can be clearly seen from Figure 5.15. This is due to bytecode modifications (e.g., renaming classes, fields, and variables, package hierarchy flattening, class repackaging, merging classes and modifying package access permissions) performed by ProGuard combined with a decompilation process that greatly changed both the lexemes and the structure of the code. It is interesting to see that jaccard and sorensen-dice, an n -gram matching technique, are the highest performing tools with F-scores of 0.5000 and 0.5287 for P_gK and P_gP_c respectively. Thus, in the presence of pervasive modifications that heavily or completely change code structure, using a simpler, general, text similarity technique may give a higher chance of finding similar code than dedicated code similarity detection tools.

Code after source code obfuscation by Artifice and decompilation by Krakatau and Procyon (AK , AP_c) has comparable results to K and P_c with marginal differences. Fuzzywuzzy and jplag-java are the best tools for this modification type.

Lastly, two combinations of obfuscation and decompilation (AP_gK , AP_gP_c) also provide almost identical F-score results to P_gK and P_gP_c . This suggests that the pervasive modifications made to source code obfuscation may be no longer effective if decompilation is included. Vice versa, the modifications made by bytecode obfuscation persist through the compilation and decompilation process. Sorensen-dice and jaccard are the best tools for this modification type.

To sum up, we found that most of the tools perform well on detecting boiler-plate code, and report lower performance when adding pervasive modifications. Some clone detection tools can tolerate pervasive modifications made by source code obfuscators, but all are susceptible to pervasive changes made by decompilers or a combination of a bytecode obfuscator and decompilers. Plagiarism detectors offer decent results over the 10 modification types. Interestingly, token and n -gram matching techniques including fuzzywuzzy, difflib, jaccard, and sorensen-dice outperformed dedicated tools on heavily modified code with a combination of obfuscators and decompilers.

5.7.7 Overall discussions

In summary, we have answered the six research questions after performing five experimental scenarios using the OCD framework. We found that the state-of-the-art code similarity analysers perform differently on pervasively modified code. Properly configured, a well known and often used clone detector, ccfx, performed the best, closely followed by an n -gram similarity algorithm, jaccard. A comparison of the tools on boiler-plate code in the SOCO data set found the jplag-text plagiarism detector performed the best followed by simjava, simian, jplag-java, and deckard.

5.7.7.1 Lessons Learned

1. The experiment using compilation/decompilation for normalisation showed that compilation/decompilation is effective and improves similarity detection techniques with statistical significance. Therefore, future implementations of clone or plagiarism detection tools or other similarity detection approaches could consider using compilation/decompilation for normalisation.

2. Every technique and tool turned out to be extremely sensitive to its own configurations consisting of several parameter settings and a similarity threshold. Moreover, for some tools the optimal configurations turned out to be very different to the default configuration, showing one cannot just reuse (default) configurations.

Finding an optimal configuration is naturally biased by the particular data set. One cannot get optimal results from tools by directly applying the optimal derived parameter settings and similarity thresholds for one data set to another data set. The SOCO data set, where we have applied the optimal configurations from the OCD data set, clearly shows that configurations that work well with a specific data set may not be guaranteed to work with future data sets. Researchers have to consider this limitation every time when they use similarity detection techniques in their studies.

3. The chosen similarity threshold has the strongest impact on the results of similarity detection. We have investigated the use of three information retrieval error measures, precision at n , r -precision, and mean average precision, to remove the threshold completely and rely only on the ranked pairs. These three error measures are often used in information retrieval research but are rarely seen in code similarity measurements such as code clone or plagiarism detection. Using the three measures, we can see how successful the different techniques and tools are in distinguishing similar code from dissimilar code based on ranked results. The tool rankings can be used as guidelines to select tools in real-world scenarios of similar code search or code plagiarism detection, for example, when one is interested in looking at only the top n most similar source code pairs due to limited time for manual inspection or when one uses a file to query for the other most similar files.
4. Lastly, we compare the tools on a data set of pervasively modified boiler-plate code. We found that while most tools offered high performance on boiler-plate code, they performed much worse after pervasive modifications were

applied. We observed that pervasively modified code with changes made from a combination of bytecode obfuscation by ProGuard and the two decompilers had strongest effects on the tools' F-scores.

5.7.7.2 Observations for The Design of a Clone Search Approach

From the empirical study of 34 code similarity analysers on both the generated OCD data set and the SOCO data set, we made three observations that are useful for developing our code clone search approach.

1. Detecting code similarity based on n -grams of code tokens is an effective approach for pervasively modified code as shown in the results of RQ1. The two n -gram based techniques, Jaccard and Sorensen-Dice, obtained the 2nd and the 3rd rank behind CCFinderX and outperformed the other 31 tools. The same observation was found for the query-based scenario in RQ5. The two tools were ranked the 2nd and the 3rd on precision at n, ARP, and MAP on the OCD data set. So, they are both suitable for pair-based detection of similar code, such as in clone or plagiarism detection, and query-based detection of similar code, such as code clone search.
2. Besides Jaccard and Sorensen-Dice, two token-based Python's FuzzyWuzzy and Difflib libraries also performed well on the two data sets. They were ranked the 4th and the 6th respectively on OCD data set and ranked the 8th and the 6th on SOCO data set. The major benefit of using general string similarity techniques is that they can tolerate incomplete code snippets, which is often found in online code snippets on Q&A websites.
3. We observed that using compilation/decompilation enhanced the performance of code similarity detection. This enabled us to pursue a detailed investigation of this method in the next chapter.

5.8 Threats to Validity

There are some potential threats to validity in this chapter. We separately discuss them in three aspects: construct, internal, and external validity.

5.8.1 Construct validity

We carefully chose the data sets for our experiment. We created the first data set (generated) by ourselves to obtain the ground truth for positive and negative results. We investigated whether our obfuscators (Artifice and ProGuard), compiler (javac) and decompilers (Krakatau and Procyon) offer code modifications that are commonly found in code cloning and code plagiarism (see Table 5.1). However, they may not totally represent all possible pervasive modifications found in software. The SOCO data set has been used in a competition for detecting reused code and a careful manual investigation has revealed errors in the provided ground truth that have been corrected.

5.8.2 Internal Validity

Although we have attempted to use the tools with their best parameter settings, we cannot guarantee that we have done so successfully and it may be possible that the poor performance of some detectors is due to wrong usage as opposed to the techniques used in the detector. Moreover, in this study we tried to compare the tools' performances based on several standard measurements of precision, recall, accuracy, F-score, AUC, prec@n, ARP and MAP. However, there might be some situations where other measurements (e.g., Matthews correlation coefficient or normalised discounted cumulative gain) are required and that might produce different results.

5.8.3 External Validity

The tools used in this study were restricted to be open-source or at least be freely available, but they do cover several areas of similarity detection (including string-, token-, and tree-based approaches) and some of them are well-known similarity measurement techniques used in other areas such as normalised compression distance (information theory) and cosine similarity (information retrieval). Nevertheless, they might not be completely representative of all available techniques and tools.

The generated OCD (100 Java files) and SOCO (259 Java files) data sets are

fairly small and contain a single class with one or a few methods. They might not adequately represent real software projects. Hence, our results are limited to pervasive modifications and boiler-plate code at a file-level, not a whole software project. The optimal configurations presented in this paper are found relative to the data set of code modifications from which they were derived and may not generalise to all types of code modifications. In addition, the two decompilers (Krakatau, Procyon) are only a subset of all decompilers available. So they may not totally represent the performance of the other decompilers in the market or even other source code normalisation techniques. However, we have chosen two instead of only one so we can compare their behaviours and performances. As we are exploiting features of Java source and byte code, our findings only apply to Java code.

5.9 Related Work

There are a few frameworks and data sets for evaluating code clone and plagiarism detectors. Nevertheless, creating a good data set for code similarity evaluation is challenging. Here we discuss the existing framework for code similarity detectors and discuss their strengths and weaknesses compared to our OCD framework.

Bellon et al. [2007] manually classified 2 percent of 325,935 clone candidates from eight subject systems in C and Java reported by six clone detectors. Since the clone ground truth comprises of the 2-percent manually validated clone pairs, the measure of precision gives only the lower bound.

Roy and Cordy [2009a] creates a mutation/injection-based automatic framework for evaluating code clone detection tools by applying mutation operators to create clones. The framework imitates code changes made to clones of Type-1 to Type-3 but does not include disruptive changes such as code rewriting, i.e., Type-4 clones. Their framework is mostly limited to locally confined modifications, only including systematic renaming as a pervasive modification. Due to the limitations, we have not included their framework in our study. We rely on code obfuscators to make locally confined modifications similar to their framework. Moreover, we

apply compilation and decompilation to create semantically similar code. However, since our OCD framework apply multiple code modifications on top of each other, we cannot precisely measure precision and recall on a specific clone type as Roy’s mutation framework.

Svajlenko et al. create a large-scale clone benchmark containing more than eight million clone pairs mined from IJaDataset, the repository of 25,000 Java open source projects by using regular expressions of 43 base functionalities [Svajlenko and Roy, 2016]. The size of the benchmark is suitable for measuring scalability of clone detectors. However, it is not suitable for measuring precision since only a partial clone ground truth is manually validated based on the 43 base functionalities. Moreover, measuring of recall is only based on the known clone pairs in the ground truth. Although our two data sets in this study are much smaller in size in comparison with the BigCloneBench, we were able to measure both precision and recall. Since we created one data set using code obfuscators, a compiler, and decompilers, and reused another data set from a competition, we had a complete knowledge of the ground truth for both of them and could take all possible similar code pairs, i.e., clones, into account.

Several code obfuscation methods can be found in the work of Luo et al. [Luo et al., 2014]. The techniques utilised include obfuscation by different compiler optimization levels or using different compilers. Obfuscating tools exist at either source code level (e.g., Semantic Designs Inc.’s C obfuscator, Stunnix’s CXX-obfuscator), and binary level (e.g., Diablo, Loco [Madou et al., 2006], CIL [Necula et al., 2002]). Their study is based on C programs while our study is based on Java. Similarly, we employed both source-level (Artifice) and bytecode-level (ProGuard) Java obfuscators in this study.

An evaluation of code obfuscation techniques has been performed by Ceccato et al. [2009]. They evaluated how layout obfuscation by identifier renaming affects the participants’ comprehension of, and ability to modify, two given programs. They found that obfuscation by identifier renaming could slow down an attack by two to four times the time needed for clear, un-obfuscated programs. Their later

study [Ceccato et al., 2013] confirms that identifier renaming is an effective obfuscation technique, even better than control-flow obfuscation by opaque predicates. Our two chosen obfuscators also perform layout obfuscation, including identifier renaming, in this study. However, instead of measuring understanding of obfuscated programs by human, we measure how well code similarity analysers perform on obfuscated code, which we use as a kind of pervasive code modifications. We also decompiled obfuscated bytecode and compared the tools' performances based on the resulting source code.

Keivanloo et al. [2015] discussed the problem of using a single threshold for clone detection over several repositories and propose a solution using threshold-free clone detection based on unsupervised learning. The method mainly utilises k -means clustering with the Friedman quality optimization method. Our investigation of precision at n, ARP, and MAP focuses on the same problem but our goal is to compare the performance of several similarity detection tools instead of boosting the performance of one tool as in their study.

The work that is closest to ours is the empirical study of the efficiency of current detection tools against code obfuscation [Schulze and Meyer, 2013]. The authors created the Artifice source code obfuscator and measured the effects of obfuscation on clone detectors. However, the number of tools chosen for the study was limited to only three detectors: JPlag, CloneDigger, and Scorpio. Nor has bytecode obfuscation been considered. The study showed that token-based clone detection outperformed text-, tree- and graph-based clone detection (similar to our findings).

5.10 Chapter Summary

This chapter presents the OCD framework for evaluating code similarity analysers and a broad empirical study of 34 tools using the framework. Our experimental results show that highly specialised source code similarity detection techniques and tools can perform better than more general textual similarity measures. However, general string matching techniques, jacard, sorensen-dice, fuzzywuzzy and difflib,

outperform dedicated code similarity detection tools in some cases especially for code with heavy structural changes. The results of the empirical study can be used as a guideline for researchers to select a proper technique with appropriate configurations for their data sets and also to compare future tools based on the existing results presented in this chapter.

The OCD framework, which we introduce in this chapter, will be used to evaluate our scalable code clone search tool in Chapter 7 and compare to the results of other tools found in the empirical study in this chapter. A few observations we made from the empirical study will also enable us to choose an appropriate code similarity technique for the clone search approach.

Moreover, we confirmed that compilation and decompilation can be used as an effective normalisation method that greatly improves similarity detection between Java source code, leading to three clone and plagiarism tools not reporting any false classification on our OCD data set. The next chapter will pursue a further investigation of using compilation and decompilation to enhance code clone detection. It applies the same technique introduced in this chapter to three real-world Java projects and analysed the detected clone pairs.

Chapter 6

Using Compilation/Decompilation to Enhance Code Clone Detection

This chapter is a follow-up to the findings in the previous chapter by studying the effects of compilation and decompilation to code clone detection in more detail. As previously observed, compilation/decompilation canonicalise syntactic changes made to source code and can be used as a source code normalisation technique. This chapter will apply the technique to a software project, instead of a file as previously done, and evaluate its effectiveness in increasing recall of a clone detector.

6.1 Motivation

We aim to exploit compilation and decompilation as a pre-processing step for detecting clones in Java programs. The previous chapter has shown that compilation/decompilation can enhance the performance of 34 code similarity analysers, including clone detection tools. This is because the process of compilation and decompilation canonicalise differences between source code files and can be considered as a code normalisation technique. Similar work is detecting clones after compilation within Jimple code [Selim et al., 2010], bytecode [Chen et al., 2014, Kononenko et al., 2014], or assembler code [Davis and Godfrey, 2010]. However, instead of doing clone detection at an intermediate level such as bytecode, Jimple, or assembler level, we use decompilation into Java source code to be able to use **any** Java source code clone detector.

Detecting clones after compilation/decompilation has three major benefits. First, code decompilation generates a second set of source code that can be useful for manual investigation of clones. In this chapter, we find that some clones discovered after decompilation are interesting and sometimes can be used as a recommendation for code refactoring. This insight cannot be achieved by looking at clones at bytecode or assembler code level. Second, it supports existing state-of-the-art clone detection tools. Since the decompiled code is Java source code, one can choose any available Java clone detector. Third, performing clone detection after decompilation can also be used in a case in which access to the source code is not available or restricted.

While using compilation/decompilation to augment clone detection has shown promising results, the dataset used in Chapter 5 was limited to 10 small Java programs. They do not represent a real environment in software systems with hundreds or thousands of source code files with third-party APIs and dependencies among classes.

This chapter performs clone detection on three real-world software systems and compares the results before and after decompilation. We resort to the build mechanism provided in each project to handle dependencies in the compilation process and use a decompiler to retrieve a decompiled versions from the class files. The findings show that using compilation/decompilation to enhance clone detection can be applied to real-world software systems. Furthermore, there are clones that are challenging to detect in the original code but can be discovered after decompilation (see Figure 6.3, Figure 6.4 and Figure 6.5 for examples). This opens a possibility of using decompilation to increase recall of clone detectors.

6.2 Contributions

This chapter makes the following primary contributions:

1. **A study of effects of compilation/decompilation to clone detection:** We demonstrate that using compilation/decompilation as a pre-processing step of clone detection is feasible for real-world Java projects. By combining clones

found before and after decompilation, one can achieve higher recall without losing precision.

2. **Providing insights into decompiled clones:** Our manual investigation shows that there are clones which can only be discovered using compilation and decompilation. We summarise their characteristics.
3. **Clone oracle:** 326 manually validated clone pairs can be used as a clone oracle in future clone studies.

6.3 Experimental Design

The study aimed to answer the following research questions:

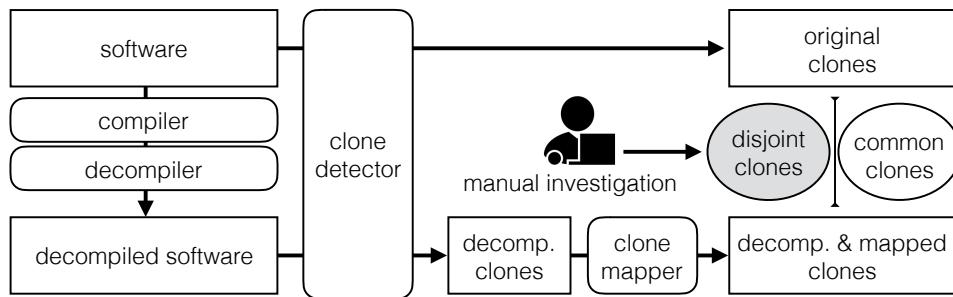
RQ1 (Clone agreement): *How many clone pairs are mutually agreed and reported by the same clone detector before and after decompilation?*

RQ2 (Decompilation accuracy): *How does compilation/decompilation affect precision and recall of clone detectors?*

RQ3 (Characteristics of disjoint clones): *What are the characteristics of clones discovered only in the original source code before decompilation and only in the decompiled source code?*

6.3.1 Experimental Framework

The framework of the study is depicted in Figure 6.1. Given a software system, we remove comments and apply pretty-printing to the source code. The system is then compiled and decompiled to generate another version of the software. A clone detector is applied to both versions. This process generates two clone reports: one for the original code and another one for the decompiled code. We are interested in method-level clones in this study, so the clone report contains file names, starting lines and ending lines of cloned method pairs. Since starting and ending line of the clones in the decompiled clone report are different from the original report, we cannot compare the decompiled clones to the original clones directly. Thus, we

**Figure 6.1:** The experimental framework**Table 6.1:** Software systems

System	Version	Original		Decompiled	
		Files	SLOC	Files	SLOC
JUnit	4.13	203	9,777	311	11,233
JFreeChart	1.5.0	644	96,711	669	85,251
Tomcat	9.0	1,688	241,924	2603	256,974

build a mapping tool to map the starting and ending lines of decompiled clones to their respective locations in the original code and generate another version of the report, *decompiled-and-mapped* clone report. We compare the original and decompiled-and-mapped clone report to find common and disjoint clone pairs. Finally, we manually look at the disjoint pairs to check if they are true clones.

6.3.2 Software Systems

We select the latest versions (obtained on 19 November 2016) of three well-known Java open source systems for this study: *JUnit v.4.13*, *JFreeChart v.1.5.0*, and *Apache Tomcat v.9.0* from GitHub. The size¹ of three systems are varied as listed in Table 6.1. Tomcat is the largest project in the set having approximately 240K SLOC. It is 2.5 times bigger than JFreeChart and 25 times bigger than JUnit. We are only interested in Java production source code but not test code, so we remove all testing class files before the analysis.

6.3.3 Tools

The following tools are used for this study.

¹The size is measured in terms of SLOC (excluding comments and blank lines) by cloc tool (<https://github.com/AIDanial/cloc>)

Table 6.2: NiCad’s configurations

Configuration	Parameters
Type-1	UPI=0.0, renaming=none
Type-2	UPI=0.0, renaming=consistent
Type-3	UPI=0.3, renaming=consistent

6.3.3.1 Compiler and Decompile

We use the standard *javac* as the compiler and an open-source tool *Procyon* [Strobel, 2015] as the decompiler². Procyon has advantages over other Java decompilers for its ability to handle declaration of `enum`, `String`, `switch` statements, anonymous and named local classes, annotations, and method references. Moreover, it is shown in the previous chapter that Procyon produces decompiled code that is easier to read than Krakatau.

6.3.3.2 Clone Detector and Its Configurations

We select the well-known NiCad tool as the clone detector for this study. NiCad has been used extensively in several clone research studies [Ragkhitwetsagul et al., 2016a, Svajlenko et al., 2014b, Wang et al., 2013b, Sajnani et al., 2016]. It can detect clones at method level which suitably supports our clones mapping algorithm. An additional benefit of using NiCad is its ability to detect and categorise clones into Type-1, Type-2, and Type-3 by choosing from its pre-defined configuration files. We select three sets of parameter configurations for NiCad as listed in Table 6.2. The default configuration (UPI=0.3, renaming=none) does not conform to any clone type and is also subsumed by the Type-3 configuration, so we do not include it in this study³. Our method allows other method-level clone detectors such as DECKARD, or SourcererCC to be used if required. However, in this chapter, we focus more on the effects of decompilation to different clone types rather than comparing different tools and detection approaches.

²We have also tried Krakatau but it failed to decompiled many of the class files so we did not adopt it in this study.

³Although the thesis has shown in the previous chapter that using the default configurations may not give optimal performance, we could not tune the parameters of NiCad for this data set because they do not provide the clone ground truth.

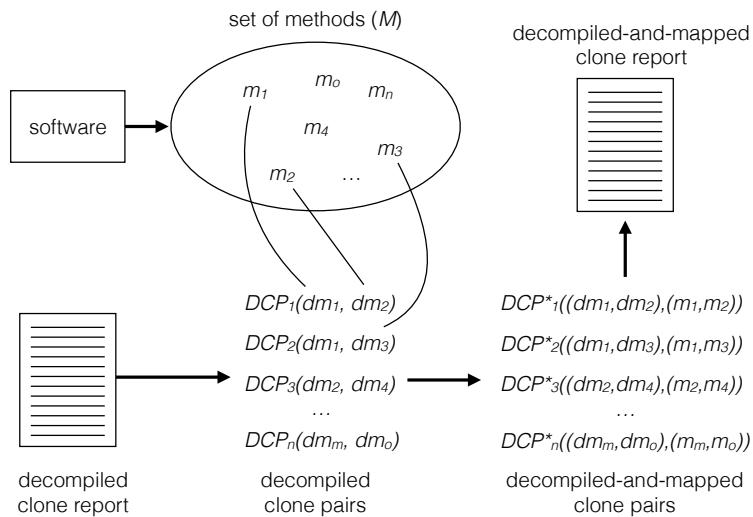


Figure 6.2: The process of mapping decompiled clones to their original locations

6.3.3.3 Clone Mapping Tool

In a regular clone detection activity, one runs a clone detection tool against a software system or multiple software systems and consults a clone report to locate clones in the software. In this study, we have not only an original software system but also another decompiled version of the software. We implemented a clone mapping tool that automatically processes decompiled clone pairs and maps them back to their original locations. The tool offers several benefits. With the clone mapper, we can compare clones before and after decompilation just by using line numbers. Moreover, after mapping, one can directly incorporate decompiled clones into their original results since their locations are consistent with the original code. Finally, the generated clone report conforms to the format of NiCad clone report and can be analysed by other clone evaluation frameworks based on clone lines (e.g., Bellon's [Bellon et al., 2007], BigCloneBench [Svajlenko and Roy, 2015], EvaClone [Wang et al., 2013b]) in the same way as the original.

The overview of the clone mapping process is shown in Figure 6.2. The tool works at method level. The clone mapping algorithm relies on a fully-qualified class name, method name, and its parameters as matching criteria. The clone mapper tool starts by extracting a set M of all methods and constructors from a software system under analysis. A method x is stored as a vector m_x containing

a method name, vector p of parameters, starting line, ending line, and fully-qualified class name: $m_x = [name, p, start, end, FQClassName]$. Then, the tool reads a decompiled clone report and extracts all decompiled clone pairs (DCP). Each DCP contains two decompiled methods $DCP(dm_x, dm_y)$ reported as clones to each other. Clone mapper iterates over all decompiled clone pairs and tries to match each decompiled method to every original method in M based on $name$, p , and $FQClassName$ by string matching. For example, as illustrated in Figure 6.2, a decompiled clone pair $DCP_1(dm_1, dm_2)$, finds matches between dm_1 and m_1 and between dm_2 and m_2 . Then, the clone mapper creates a decompiled-and-mapped clone pair $DCP^*_1((dm_1, dm_2), (m_1, m_2))$ containing the clone pair with locations in both decompiled and original source code. If there is no match, that means the matching method does not exist in the original source code and is solely generated by the process of compilation and decompilation (for example, default constructors). The tool ignores such unmatched methods and all its respective clone pairs. After all the decompiled clone pairs are processed, the clone mapper generates a *decompiled-and-mapped* clone report from the set of DCP^* . The decompiled-and-mapped clone report is used along with the original clone report to find common and disjoint clone pairs.

6.3.3.4 Common and Disjoint Clone Pairs

Using the original and decompiled-and-mapped clone report, we extract two sets of clone pairs: C_{orig} and C_{decomp} . We find clone pairs that are common between them by performing a set intersection. We call clone pairs in the intersection *common clone pairs* (C_{common}).

$$C_{common} = C_{orig} \cap C_{decomp}$$

Clone pairs that can only be found in the original ($C_{orig-only}$) and decompiled set ($C_{decomp-only}$) are results of subtraction by the common clone pairs. We call them *disjoint clone pairs*.

$$C_{orig-only} = C_{orig} - C_{common}$$

$$C_{decomp-only} = C_{decomp} - C_{common}$$

We mainly focus on disjoint clone pairs for manual investigation. This approach gives us clones that are detected only before and after decompilation. By focusing on the disjoint clone pairs, we can reduce the number of clones that need to be manually investigated dramatically and can study them in more details.

6.3.3.5 Clone Filtering

Before starting the manual clone validation process of the three systems, the thesis author sampled a few clone pairs to look at manually and found some trivial and auto-generated clone pairs. They were `equals()`, `hashCode()`, getters, setters, and duplicated methods generated by the compiler and decompiler. We filter such clone pairs using regular expressions because they are not very interesting to look at. The `equals()`, `hashCode()`, getter and setter clone pairs are similar boiler-plate code. The duplicated methods are inner-class methods which are by-products from the compilation/decompilation process. They must be removed since they do not exist in the original code.

6.4 Results and Discussion

We performed an experiment on an Apple iMac machine running macOS 10.12.1 with 2.7 GHz Intel Core i5 and 8 GB of RAM. The answers to the three research questions are discussed below.

6.4.1 RQ1: Clone Agreement

How many clone pairs are mutually agreed and reported by the same clone detector before and after decompilation?

We answer RQ1 by running NiCad against the three software systems twice, before and after decompilation, and studying the clones. NiCad was configured using three different configurations: Type-1, Type-2 with consistent renaming, and Type-3 with consistent renaming (i.e., using NiCad's configuration file `type1.cfg`, `type2c.cfg`, and `type3-2c.cfg` accordingly). NiCad provided blind and consistent renaming options. We chose the stricter consistent renaming so that we could reduce the number of false positives. Then, we used the clone mapper to map decompiled clones back

to their original counterparts. Finally, we computed an intersection of clone pairs between the *original* (C_{orig}) and *decompiled* (C_{decomp}) set to find common and disjoint clone pairs. The same approach of finding common and disjoint clones before manual analysis has also been done by Kononenko et al. [2014].

The number of clone pairs in common, orig-only, and decomp-only set before and after filtering are displayed in Table 6.3. The set of clone pairs after filtering is denoted as C_f . The clones are divided by clone types from Type-1 to Type-3. The numbers are mutually exclusive. For example, Type-2 clone pairs are pairs that are found using Type-2 configurations and not reported in Type-1 pairs. Similarly, the number of Type-3 clone pairs are the ones not reported in Type-1 and Type-2. The findings from the three systems are discussed below.

6.4.1.1 JUnit

The system contained no Type-1 clone. After filtering, we found 6 Type-2 and 3 Type-3 clone pairs and all of them were identically reported from both before and after decompilation. We did not find any disjoint clone pairs, so we did not continue the manual investigation for JUnit.

6.4.1.2 JFreeChart

The followings are numbers after filtering. For Type-1 clones, we found 33 (89.2%) common, 1 (2.7%) orig-only, and 3 (8.1%) decomp-only pairs. For Type-2, there were 159 (83.2%) common, 15 (7.9%) orig-only, and 17 (8.9%) decomp-only clone pairs. For Type-3, there were 155 (67.4%) common, 48 (20.9%) orig-only, and 27 (11.7%) decomp-only pairs.

6.4.1.3 Tomcat

After filtering, we found 20 (46.5%) common, 22 (51.2%) orig-only, and 1 (2.3%) decomp-only clone pairs in Type-1. For Type-2, there were 217 (88.6%) common, 25 (10.2%) orig-only, 3 (1.2%) decomp-only clone pairs. Lastly, for Type-3, there were 608 (78.8%) common, 141 (18.3%) orig-only, 23 (2.9%) decomp-only pairs.

Table 6.3: Systems and clones found categorised by clone types. The numbers are clone pairs found only in a particular clone type (non-subsuming). C denotes “clone pairs” and C_f denotes “filtered clone pairs”.

System	Clone type	$ C_{common} $	$ C_{orig-only} $	$ C_{decomp-only} $	$ C_{f common} $	%	$ C_{f orig-only} $	%	$ C_{f decomp-only} $	%
JUnit	Type-1	0	0	11	0	0.0	0	0.0	0	0.0
	Type-2	6	0	0	6	100.0	0	0.0	0	0.0
	Type-3	4	0	0	3	100.0	0	0.0	0	0.0
JFreeChart	Type-1	43	1	10	33	89.2	1	2.7	3	8.1
	Type-2	535	42	40	159	83.2	15	7.9	17	8.9
	Type-3	25604	12006	1885	155	67.4	48	20.9	27	11.7
Tomcat	Type-1	24	27	254	20	46.5	22	51.2	1	2.3
	Type-2	270	34	7	217	88.6	25	10.2	3	1.2
	Type-3	790	161	121	608	78.8	141	18.3	23	2.9
Total		27276	12271	2328	1201	78.7	252	16.5	74	4.8

To answer RQ1, we found that, after filtering irrelevant clone pairs, the clone pairs before and after decompilation were mostly similar for all three clone types. In JUnit, 100% of clone pairs were identically reported before and after decompilation. In JFreeChart and Tomcat, common clone pairs accounted for 67.4% to 89.2%, and 45.5% to 88.6% respectively. Nevertheless, we still found a number of disjoint clones for all three clone types, i.e., there were clones that could avoid the detection before and after decompilation. The number of decomp-only clone pairs in JFreeChart and Tomcat kept increasing from Type-1 to Type-3. This demonstrates that compilation/decompilation is useful in discovering clones with changes (i.e., Type-2 and Type-3). However, it can only marginally improve the detection of Type-1 clones since they are already handled by NiCad pretty-printing.

6.4.2 RQ2: Decompilation Accuracy

How does compilation/decompilation affect precision and recall of clone detectors?

We manually investigate 326 clone pairs (252 from $Cf_{orig-only}$ and 74 from $Cf_{decomp-only}$) in JFreeChart and Tomcat. The author of the thesis took a role of an investigator. The investigator looked at the clones in the two sets and classified them as either true or false positive. For each clone pair, he checked them both in the original and the decompiled version. However, the classification was only based on the original code. He also noted the details of the clones before and after decompilation and the reason of why they were reported in only a single set. The manual investigation results are shown in Table 6.4. We can see that every clone pair, both in the original and the decompiled set, is classified as true positive except for a single one in JFreeChart orig-only Type-3 clones.

Considering the number of clones and true positive pairs in both $Cf_{orig-only}$ and $Cf_{decomp-only}$ set, we can see that NiCad offers perfect precision almost in every setting. However, regarding recall, NiCad misses a considerable amount of clone pairs that are reported only in the original or decompiled version.

Table 6.4: Manual investigation results of clone pair candidates reported in $Cf_{orig-only}$ and $Cf_{decomp-only}$

System	Type	$Cf_{orig-only}$		$Cf_{decomp-only}$	
		Candidates	True Positives	Candidates	True Positives
JFreeChart	Type-1	1	1	3	3
	Type-2	15	15	17	17
	Type-3	48	47	27	27
	<i>Sum</i>	64	63	47	47
Tomcat	Type-1	22	22	1	1
	Type-2	25	25	3	3
	Type-3	141	141	23	23
	<i>Sum</i>	188	188	27	27

6.4.2.1 JFreeChart

There are 47 true clone pairs from $Cf_{decomp-only}$ that were not found in the original version. On the contrary, there were 63 true clone pairs from $Cf_{orig-only}$ that were not reported in the decompiled version.

6.4.2.2 Tomcat

There were 27 true clone pairs from $Cf_{decomp-only}$ that were discovered after decompilation. On the other hand, 188 true clone pairs in $Cf_{orig-only}$ were missing after decompilation.

To answer RQ2, we find that original and decompiled source code do not have perfect clone recall. However, one can complement the original clone results by incorporating clones after decompilation. From the manual investigation, we find that all decompiled clone pairs are true positives. Combining two clone sets will increase recall of the tool without losing precision.

6.4.3 RQ3: Characteristics of Disjoint Clones

What are the characteristics of clones discovered only in the original source code before decompilation?

The manual investigation reveals 7 characteristics of disjoint clones from JFreeChart and Tomcat. The details of disjoint clone characteristics are described

Table 6.5: Characteristics of disjoint clones reported in $Cf_{orig-only}$ and $Cf_{decomp-only}$

Clone set	Why do they only appear in this set?	JFreeChart			Tomcat			Total
		T1	T2	T3	T1	T2	T3	
$Cf_{orig-only}$	Too small after decompilation	1	9	32	1	6	120	169
	Too different after decompilation	0	6	11	21	0	0	38
	Smaller after decompilation causing higher dissimilarity	0	0	0	0	0	5	5
	Unknown	0	0	5	0	19	16	40
$Cf_{decomp-only}$	Originals have added/deleted statements, type casts, package names	3	5	8	2	0	1	19
	Originals have different if-else	0	12	3	0	0	0	15
	Originals use different loops (for vs. while)	0	0	4	0	0	0	4
	Originals are inner-class methods	0	0	0	0	0	2	2
	Unknown	0	0	12	0	3	20	35

in Table 6.5. Three characteristics are found from clones in $Cf_{orig-only}$ and four are found from clones in $Cf_{decomp-only}$.

6.4.3.1 Disjoint Clones in $Cf_{orig-only}$

The majority of the clone pairs here did not have their counterparts after decompilation due to effects of the decompilation process. The most common characteristic is smaller clone size after decompilation. 169 pairs of the original clones were smaller after decompilation. They were smaller than the 10-line minimum clone size of NiCad and hence not reported, which made them appear only in the original set. The second characteristic is that clones become more different after decompilation (38 pairs). For example, two methods in the original source code contained a string constant with the same variable name but different values. The variables were declared outside of the clone region thus they formed an identical Type-1 clone pair. After decompilation, the constant variables had been replaced by the actual value of string literals. This made decompiled code no longer an identical clone pair. Another characteristic, observed from 5 clone pairs, is a decrease of similarity due to smaller clone size after decompilation. In some cases, a Type-3 clone pair with added lines became smaller after decompilation. The added lines were preserved while other statements were compressed or removed. Thus, the decompiled clone pair had a lower similarity value. The remaining 40 disjoint pairs did not have any

noticeable characteristics (categorised as *Unknown*).

6.4.3.2 Disjoint Clones in $Cf_{decomp-only}$

Most of the clone pairs are challenging Type-2 and Type-3 clones for NiCad. There were 19 clone pairs that the original code contained added/deleted statements, extra type castings (e.g., `(CategoryAxis)this.domainAxes.get(index)` vs. `this.rangeAxes.get(index)`), or package names in front of class names (e.g., `Map.Entry` vs. `Entry`). The added/deleted statements lowered clone similarity while extra type casts and package names affected Type-1 and Type-2 detection. These inconsistencies were standardised and the clone pairs were more similar after decompilation. Moreover, we observed 15 clone pairs having different `if-else` statements similar to the example depicted in Figure 6.3. The method `findDomainBounds()` and `findRangeBounds()` formed a Type-3 clone pair with flipped but equivalent `if-else` conditions. These `if-else` statements were canonicalised by the decompilation process and became identical. Interestingly, this Type-3 clone pair could be discovered using even stricter Type-2 configurations after decompilation. There were 4 Type-3 clone pairs with different loops, `for` and `while`. An example is shown in Figure 6.5. They turned almost identical after decompilation by having only `for` loops. Lastly, we found 2 clone pairs residing in inner classes. They were missing from the original clone set possibly due to complications in parsing. Compilation/decompilation extracted inner classes out as separated files so they were detected. There were 35 pairs only found after decompilation but without any observable characteristic (categorised as *Unknown*).

To answer RQ3, we derive seven characteristics of disjoint clones that make them discoverable only before and after decompilation. We observe that the majority of clones are reported only in the original set because of their smaller size after decompilation. The decompiled clones are still clones, but they are too small to be reported, which is a weakness in our technique. On the contrary, the characteristics of clone pairs only found by decompilation involve Type-2 and Type-3 clones with strong modifications at the syntactic level. After compilation/decompilation, the modifications are canonicalised.

```

/* original code */
@Override
public Range findDomainBounds(XYDataset dataset) {
    if (dataset==null) {
        return null;
    }
    Range r=datasetUtilities.findDomainBounds(dataset, false);
    if (r==null) {
        return null;
    }
    return new Range (r.getLowerBound() +this.xOffset,
                      r.getUpperBound() +this.blockWidth+this.xOffset);
}

/* decompiled code */
@Override
public Range findDomainBounds(final XYDataset dataset)
{
    if (dataset==null ) {
        return null;
    }
    final Range r=datasetUtilities.findDomainBounds
        (dataset, false);
    if (r==null) {
        return null;
    }
    return new Range(r.getLowerBound() +this.xOffset,
                      r.getUpperBound() +this.blockWidth+this.xOffset);
}

```

Figure 6.3: Example of type-3 clones in *findDomainBounds()* and *findRangeBounds()* that can be detected with type-2 configuration after decompilation

```

/* original code */
public void clearDomainMarkers() {
    if (this.backgroundRangeMarkers != null) {
        Set<Integer> keys=this;
        backgroundRangeMarkers.keySet();
        for (Integer key:keys) {
            clearRangeMarkers (key);
        }
        this.backgroundRangeMarkers.clear();
    }
    if (this.foregroundRangeMarkers != null) {
        Set<Integer> keys=this;
        foregroundRangeMarkers.keySet();
        for (Integer key:keys) {
            clearRangeMarkers(key);
        }
        this.foregroundRangeMarkers.clear();
    }
    fireChangeEvent();
}
}

/* original code */
public void clearRangeMarkers() {
    if (this.backgroundRangeMarkers != null) {
        Set keys=this;
        backgroundRangeMarkers.keySet();
        Iterator iterator = keys.iterator();
        while (iterator.hasNext()) {
            Integer key=(Integer) iterator.next();
            clearRangeMarkers (key.intValue());
        }
        this.backgroundRangeMarkers.clear();
    }
    if (this.foregroundRangeMarkers != null) {
        Set keys=this;
        foregroundRangeMarkers.keySet();
        Iterator iterator=keys.iterator();
        while (iterator.hasNext()) {
            Integer key=(Integer) iterator.next();
            clearRangeMarkers(key.intValue());
        }
        this.foregroundRangeMarkers.clear();
    }
    fireChangeEvent();
}
}

```

Figure 6.4: Example of type-3 clones with different loops in *clearDomainMarkers()* and *clearRangeMarkers()*

```

/* decompiled code */
public void clearDomainMarkers() {
    if (this.backgroundDomainMarkers!=null) {
        final Set<Integer> keys=this.
            backgroundDomainMarkers.keySet();
        for (final Integer key:keys) {
            this.clearDomainMarkers(key);
        }
        this.backgroundDomainMarkers.clear();
    }
    if (this.foregroundDomainMarkers!=null) {
        final Set<Integer> keys=this.
            foregroundDomainMarkers.keySet();
        for (final Integer key:keys) {
            this.clearDomainMarkers(key);
        }
        this.foregroundDomainMarkers.clear();
    }
    this.fireChangeEvent();
}

```

Figure 6.5: The `clearDomainMarkers()` and `clearRangeMarkers()` that can be detected after decompilation

6.5 Overall Discussion

The findings in this chapter suggest using decompilation⁴ as a complementary method to clone detection in Java software projects. Since the decompiled code is also Java source code, any source code-based clone detection tool can be benefited from this technique. We show that combining clones from before and after decompilation can increase recall without sacrificing precision. It is useful for code clone detection in a software project or between software projects.

However, the technique has a few limitations that prevent us from integrating it into a scalable clone search engine. First, a successful compilation process needs complete dependencies, so it will not work in the case of code snippets that are separated from their projects. Second, code compilation, especially in Java, relies on build platforms such as ant, gradle, or maven. These platforms manage the dependencies required for a successful compilation. To enable the technique to work on any project, we need to tailor the clone search tool to support most of, or all, of the build platforms. This hinders the generalisation of the tool. Third, compilation and decompilation process needs source code that is compilable. The technique will not work with incomplete code snippets that are usually found on Stack Overflow.

6.6 Threats to Validity

There are some potential threats to validity in this chapter. We separately discuss them in two aspects: internal and external validity.

6.6.1 Internal Validity

The three chosen software systems for our experiment might not represent all Java software projects, and the results might not be generalised. We are aware of the effects of configurations to the tools' performance, so we tuned NiCad using multiple pre-defined configurations. At the same time, they are configurations that conform to the definitions of Type-1, Type-2, and Type-3 clones. Nevertheless, we

⁴This possibly includes any kind of source-code transformation that normalizes code in the same way as compilation and decompilation.

only selected subsets of all possible NiCad configurations.

6.6.2 External Validity

All the tools used in this study are restricted to being open source to encourage replication. However, there is only a single clone detection tool and decompiler chosen. They might not represent other clone detectors and decompilers.

6.7 Related Work

There have been a few studies similar to ours by trying to detect clones after compilation. Chen et al. [2014] locate clones in Android apps based on dex files extracted from Android APKs. Davis and Godfrey [2010] convert Java and C/C++ code into assembler code and detect clones using longest common subsequence string matching augmented by hillclimbing search for flexible matching. Kononenko et al. [2014] similarly find clones in Java after compilation by adapting CCFinderX to be compatible with bytecode sequences and manually investigate disjoint clone pairs. Selim et al. [2010] enhance Simian and CCFinderX by transforming Java code into Jimple code and locating clones at that level. Their technique helps the tools to detect more Type-3 clones and handle gapped clones. Our study detect clones at source code level using the current state-of-the-art code clone detection tool after applying a two-step process of compilation and decompilation. This approach provides opportunities to compare and study clones before and after decompilation which provide several useful insights. In various cases, we find that the decompiled clones are more compact and concise than the original code.

6.8 Chapter Summary

This chapter studies compilation and decompilation as a pre-processing step for clone detection in three open source software systems. It is found that the technique can increase clone recall while not sacrificing precision. The technique is recommended for intra- or inter-project code clone detection.

The next chapter will present the architecture of our scalable and incremental clone search approach and its implementation as a tool called *Siamese*. The chapter

will evaluate the Siamese tool on multiple data sets, including the OCD framework, BigCloneBench, and GitHub, and compared the tool to the state-of-the-art clone detection tools.

Chapter 7

Siamese: Scalable and Incremental Code Clone Search Engine

This chapter sets off by explaining the architecture of Siamese, a scalable and incremental code clone search approach. It moves to discuss two main modules which enable accurate clone search: multi-representation and query reduction. Lastly, the chapter evaluates the performance of Siamese both in terms of clone search precision and scalability.

7.1 Motivation

Code search is becoming increasingly important when considering the plethora of source code currently proliferating on the Internet [Sadowski et al., 2015]. Developers prefer to reuse coding solutions from online sources, such as Stack Overflow, instead of official documentation or books [Acar et al., 2016]. Researchers have also leveraged large amounts of online code snippets to make suggestions to developers during development [Keivanloo et al., 2014, Park et al., 2014, Ponzanelli et al., 2013, 2014]. Online code snippets may be exploited for program repair [Ke et al., 2015] or code examples [Keivanloo et al., 2014, Nasehi et al., 2012]. On the other hand, reusing code from online sources have been found to introduce negative effects to software quality [Abdalkareem et al., 2017, Acar et al., 2016] or to violate software licenses [An et al., 2017, Baltes et al., 2017]. To locate such clones, a special type of code search, namely code clone search, which accepts a code

fragment as a query and performs a code-to-code search in large code corpora [Kim et al., 2018, Nishi and Damevski, 2018] is needed.

It is difficult to obtain high precision, recall, and scalability at the same time in code clone detection. Text-based search engines such as Bing and Google are scalable to the Internet but are not designed for source code and rely only on keyword search [Sadowski et al., 2015]. Dedicated code search engines such as BlackDuck OpenHub [BlackDuck, 2016], Krugle [Aragon Consulting Group, Inc., 2018] or Searchcode [Boyter, Ben, 2018] cannot efficiently handle code clones with modifications [Keivanloo et al., 2014, Kim et al., 2018]. Hummel et al. [2010] and Koschke [2014] are among the first to propose scalable clone detection systems. However, the trade-off for the scalability is their ability to report only copy-and-paste clones or clones with variable renaming (i.e., Type-1 and Type-2 clones), while the largest number of clones found in software are clones with added or deleted statements (Type-3 clones) [Roy and Cordy, 2009b, Svajlenko et al., 2014b]. Although there are scalable clone detection and clone search techniques that can locate Type-3 clones with some level of success [Keivanloo et al., 2011, Sajnani et al., 2016, Kim et al., 2018], scalably finding Type-3 clones is still an open challenge.

Retrieving a ranked list of clones is preferred over a full list of clone pairs in various contexts, such as finding similar code examples [Keivanloo et al., 2014] or searching for candidates for bug fixing [Ke et al., 2015]. Code clone detectors that report a complete set of clones are not suitable for these tasks because a large number of clone pairs have to be manually investigated [An et al., 2017, Yang et al., 2017, Bauer et al., 2016]. In these circumstances, the user would only need a ranked list of top n cloned code fragments instead [Niu et al., 2017]. There have been a number of code search tools which produce a ranked list of code candidates [Grechanik et al., 2010, Inoue et al., 2012, Keivanloo et al., 2014, Kim et al., 2018, McMillan et al., 2011, Niu et al., 2017, Zhang et al., 2017] for some specific use cases. To support a broader range of applications, we prefer a clone search engine that is general and not tied to any specific use cases or scenarios.

Moreover, to find good candidates for program repair, we look for clones which deviate from the original buggy code (i.e., Type-3/Type-4) to increase the chance of successful repairs [Ke et al., 2015]. On the other hand, to check for copy-and-paste code from online sources and investigate their license compatibility, we are interested in clones that are closer to the original (i.e., Type-1/Type-2) to reduce the manual investigation time. Thus, it is important that the clone search tool captures different types of clones.

Lastly, most of the clone detectors do not handle incremental addition or deletion of software projects. Thus, adding new projects to the code base under analysis or updating existing projects would result in the need to rerun the clone detection for the complete data set. Several of the proposed techniques that support incremental clone detection do not scale to large-scale data sets [Göde and Koschke, 2009, Kawaguchi et al., 2009, Nguyen et al., 2009b] or do not detect Type-3 clones in sacrificing for scalability [Hummel et al., 2010, Koschke, 2014].

7.2 Contributions

To tackle these challenges in large-scale clone search, we present and evaluate a scalable code clone search engine that retrieves Type-1 to Type-3 code clones in seconds, and supports incremental changes in software projects. The **Siamese** (Scalable, incremental, and multi-representation) clone search engine works with multiple representations of source code to capture code similarity at different structural levels. It mines token frequencies in a code corpus on-the-fly and automatically adjusts a query’s length to improve the search speed and accuracy. The tool allows incremental updates to its source code index. The evaluation of Siamese shows that it scales to 365M SLOC and returns the results within 10 seconds. Our technique offers a search precision of 95% and 99% on two established clone benchmarks, which are higher than seven state-of-the-art clone detection tools. Moreover, the technique also exhibits high recall and precision for all clone types in the BigCloneBench [Svajlenko et al., 2014b], a large-scale clone benchmark. This chapter makes the following primary contributions.

1. A multi-representation and query reduction techniques for code clone search that is accurate and scalable, and their evaluation.
2. The Siamese clone search engine¹ which is scalable and incremental, suitable for performing instant clone search on large-scale data sets, such as online code repositories.

7.3 Siamese Clone Search Architecture

We designed the architecture of our clone search approach by adopting inverted index and code clone detection techniques as depicted in Figure 7.1. Source code from code corpora is stored in an inverted index, which is a widely-used data structure for fast querying of relevant documents [Manning et al., 2009]. Our architecture separates the necessary indexing of source code, where the search index is created, from querying, where the clones of a queried code fragment are retrieved. Inverted index and tf-idf-based scoring functions are exploited as the infrastructure of code retrieval and similarity measurement. Siamese works at token level which supports scalable detection of near-missed clones [Kim et al., 2018, Sajnani et al., 2016]. The two techniques normally found in token-based clone detection including token normalisation [Kamiya et al., 2002, Prechelt et al., 2002, Roy and Cordy, 2008] and n -gram generation [Burrows et al., 2007, Ohmann and Rahal, 2014, Prechelt et al., 2002, Schleimer et al., 2003] are performed during indexing and querying time.

The architecture incorporates a novel multi-representation and query reduction technique to increase clone search precision and flexibility of clone matching. The **multi-representation module** (Section 7.3.3) enables clone detection based on multiple code representations instead of one representation as in other tools. The **query reduction module** (Section 7.3.4) leverages the knowledge of token document frequency in a code corpus to improve the quality of the query on-the-fly. Our **customised scoring and ranking module** (Section 7.3.5) computes scores for matched code fragments and returns a ranked list of clones. Lastly,

¹Tool and data sets used are available at <https://siamesetool.github.io/siamese>.

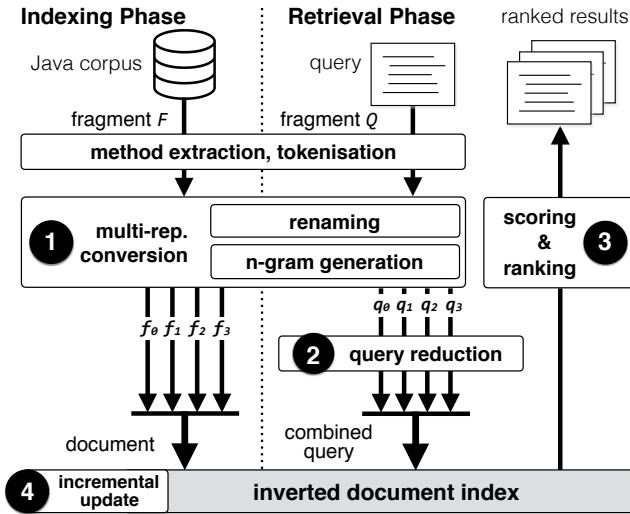


Figure 7.1: Siamese Architecture

the **incremental update module** (Section 7.3.6) allows the user to add new code fragments to the index or delete selected existing code fragments from the index without affecting other indexed code fragments. Siamese performs a two-phase approach: an indexing and a querying phase.

7.3.1 Indexing Phase

In this phase, Siamese processes a given source code base(s) to generate a searchable code index. Siamese supports two types of code fragments, files and methods, and the input code fragments are preprocessed before being stored into the inverted index. Siamese is a token-based tool and is resilient to incomplete or uncompilable code fragments. If the method parsing fails, it falls back to store the source code at a file level. Each input code fragment F (file or method depending on its granularity) is then tokenised into a stream of tokens and sent to the *multi-representation (MR)* conversion module to generate four code representations which capture the code structure at different levels, before being stored in the index. Indexing source code files is an expensive task because the tool has to process all the available code data. Fortunately, it occurs far fewer times than the querying phase.

7.3.2 Querying Phase

The querying phase happens when the clone search tool receives a code query from its user and returns clone results. Only indexed documents containing the query

terms are retrieved and ranked. Querying is the main activity for Siamese and usually occurs many more times than indexing. In this phase, the source code query is prepared in the same way as in the indexing phase by passing through method extraction and tokenisation steps. A tokenised code query Q is sent to the MR module to generate four query representations, i.e., siblings. The *query reduction (QR)* module rewrites and generates reduced queries from the original four query siblings. The reduced query siblings are combined into a single search request and executed on the search engine. Siamese retrieves indexed code fragments that match with the combined query and computes the ranking of results using a customised scoring function before reporting them to the user.

7.3.3 Multi-Representation (MR)

The Siamese clone search approach works with four code siblings derived from the original source code fragment F by the multi-representation module. The set of four code representations $\{r_0, r_1, r_2, r_3\}$ that represent F are defined as follows.

1. **Original representation r_0** : A stream of tokens, i.e., 1-grams, containing tokens from the original source code (text search).
2. **Type-1 representation r_1** : A stream of n -grams containing tokens from the original code (Type-1 clone search).
3. **Type-2 representation r_2** : A stream of n -grams containing normalised n -grams with identifier, literal, and type tokens replaced by the representative tokens (Type-2 clone search).
4. **Type-3 representation r_3** : A stream of n -grams containing normalised n -grams with all tokens replaced by the representative tokens, except Java punctuators {, }, [,], (), and ;. Punctuators are not normalised as they are meaningful to the code structure (Type-3 clone search).

The three n -gram-based representations (r_1, r_2, r_3) are derived from the stream of tokens in the original representation (r_0). Our MR module augments the normal text search and makes it more suitable for code search by including three more

Table 7.1: Representative tokens for specific token types

D	data types	J	Java class names
K	Java keywords	P	Java packages
O	operators	S	string literals
V	numbers	W	words (other identifiers)

```

public static int binarySearch1 (int arr[], int key, int imin,
    int imax) {
    if (imax < imin)
        return -1;
    int imid = (imin+imax)/2;
    if (arr[imid] > key)
        return binarySearch1(arr, key, imin, imid-1);
    else if (arr[imid] < key)
        return binarySearch1(arr, key, imid+1, imax);
    else
        return imid;
}

```

Figure 7.2: An example code fragment of a binary search method

representations that leverage token types, code structure, and the knowledge of clone types. For Type-1 representation (r_1), the n -grams are generated directly from the original representation (r_0). For Type-2 (r_2) and Type-3 representation (r_3), the stream of tokens r_0 is normalised to a reduced token stream in which tokens of specific types are replaced by a representative token. Table 7.1 shows the list of our pre-defined representative tokens containing D for data types, J for Java class names, K for Java keywords, P for Java packages, O for operators, S for string literals, V for numbers, and W for words, i.e., other identifiers. In case of r_2 , all identifiers, types, numbers, and string literals are replaced by a representative token W, D, V, and S respectively. For r_3 , all tokens are replaced with their respective representative tokens. Then, r_1 , r_2 , and r_3 are obtained by n -gramising their respective reduced token stream.

For example, given a code fragment of a binary search method in Figure 7.2, the four representations r_0, r_1, r_2, r_3 generated from the MR module are depicted in Table 7.2.

This MR technique enables Siamese to capture multiple clone types at the

Table 7.2: The four representations of the binarySearch1 method generated from the MR module.

r_0 (n -gram size = 1)
<pre>public static int binarySearch1 (int arr [] , int key , int imin , int imax) ... ; else return imid ; }</pre>
r_1 (n -gram size = 4)
<pre>publicstaticintbinarySearch1 staticintbinarySearch1(intbinarySearch1(int binarySearch1(intarr (intarr[... ;elsereturnimid elsereturnimid; returnimid;}</pre>
r_2 (n -gram size = 4)
<pre>publicstaticDW staticDW(DW(D W(DW (DW[DW[], W[], [],D],DW ,DW, DW,D W,DW ,DW) DW){ W){if){if(...);elsereturn ;elsereturnW elsereturnW; returnW;}}</pre>
r_3 (n -gram size = 4)
<pre>KKDW KDW(DW(D W(DW (DW[DW[], [],D],DW ,DW, DW,D W,DW ,DW) DW){ W){K){K({K(W K(WO (WOW ... KK(W WOW, OV,W V,W) ,W); W);K ;KKW KKW; KW;}}</pre>

same time. During the search, each code representation in the query will match with its respective representation of the indexed code fragments. We apply MR conversion to the source code in both the indexing and querying phase. In the indexing phase, Siamese creates a new document for a given code fragment and puts the four representations in separated fields inside the document. Then, the document is stored in the search index. In the querying phase, Siamese creates a combined query containing four sub queries of the four representations.

7.3.4 Query Reduction (QR)

Clone search suffers the long query problem [Kumaran and Carvalho, 2009] since a code fragment is given as a query. To tackle this problem, we adopted a query reduction technique using token document frequency (df), i.e., the number of documents in which the token appears, as a query quality predictor [Kumaran and Carvalho, 2009]. We rewrite the query to contain only rare tokens and discard frequent ones. According to studies of Zipf's power law in software [Zipf, 1932, Knuth, 1971, Zhang, 2008], there are a few highly frequent tokens in programming languages and the frequency of tokens drop rapidly inversely proportional to their

ranks. Thus rare code tokens are ranked among the last and share only a few documents with others. By choosing only rare tokens to form a reduced query, one can (1) decrease the number of retrieved code snippets to be only highly relevant ones, (2) increase the search speed due to fewer search terms to process, and (3) avoid false positive results. Our query reduction technique chooses rare tokens in a query on-the-fly by analysing df scores of all query tokens.

Siamese derives four sibling queries q_0, q_1, q_2, q_3 from the original query Q (a given code fragment), and shortens all of them. The QR module gets rid of duplicated tokens by consolidating tokens or n -grams in q_0, q_1, q_2, q_3 into a set of unique tokens and n -grams. Then it filters the tokens based on their df score. For each representation q_i , tokens with df score lower than or equal to a threshold θ_i are kept in the reduced query, and tokens with df score higher than θ_i are discarded. The θ_i value is a proportion of the number of documents in the index and can be adjusted via a variable called $dfCap_i$ (ranging from 0 to 100 percent). The threshold θ_i and each reduced token query q'_0, q'_1, q'_2 , and q'_3 are defined as below.

$$\theta_i = dfCap_i \times |documents|, i \in [0, 3]$$

$$q'_i = \{t \in q_i : df(t) \leq \theta_i\}, i \in [0, 3]$$

The optimal θ_i value for the four representations may be different based on the distribution of tokens and n -grams in each representation. Setting a low θ value offers high discriminative power since it allows only rare tokens to appear in the query, and results in a short query, while selecting a high θ_i value gives low discriminative power and allows frequent tokens to be included in the query.

7.3.5 Scoring and Ranking of the Results

Siamese exploits Apache Lucene's scoring and ranking function to create a list of ranked cloned results. The scoring and ranking technique is based on a *vector space model (VSM)* [Salton et al., 1975] representation by converting documents, i.e., code fragments, into k -dimensional weight vectors $V = \langle w_1, w_2, w_3, \dots, w_i, \dots, w_k \rangle$ where k equals the number of terms in the dictionary. A popular weighting scheme is *term*

frequency (tf) and *inverse document frequency (idf)*. *tf* represents how frequent a term occurs in a document and is defined as $\text{tf}(t,d) = \sqrt{\text{frequency}(t,d)}$. *idf* represents how often the term occurs across all the documents in the corpus and is defined as $\text{idf}(t) = 1 + \log\left(\frac{|\text{documents}|}{\text{df}(t)+1}\right)$, where $\text{df}(t)$ stands for document frequency of term t .

Apache Lucene computes a *relevance score* between a query vector and a document vector in order to gain speed in searching and ranking. Relevant documents are ranked according to their scores, i.e., their relevance to the query, before returning to the user. The Lucene scoring formula [Apache Software Foundation, 2012] is

$$\text{score}(q,d) = \sum_{t \in q} [\text{tf}(t,d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost()} \cdot \text{norm}(t,d)] \cdot \text{queryNorm}(q) \cdot \text{coord}(q,d), \quad (7.1)$$

where a $\text{score}(q,d)$ between a document d in the index and the query q is computed from a sum of term scores for all the terms in q . A score for each term t in the query is computed from a multiplication of the term frequency in document $\text{tf}(t,d)$, the squared inverse document frequencies $\text{idf}(t)^2$, the term boosting weight $t.\text{getBoost()}$, and the field length normalisation $\text{norm}(t,d)$. Finally, the sum of term scores is multiplied by a query normalisation factor, $\text{queryNorm}(q)$, and a query coordination, $\text{coord}(q,d)$ ².

Since $\text{tf}(t,d)$ will be zero for terms that do not exist in the document, only matched terms contribute to the score. Siamese relies on four representations of Java code, hence the final score of each code snippet is a sum of scores from the four reduced queries q'_0 , q'_1 , q'_2 , and q'_3 . Our customised scoring function is

$$\text{score}_{\text{Siamese}}(Q,d) = \sum_{i=0}^3 \text{score}(q'_i,d). \quad (7.2)$$

²Detailed explanation: a query normalisation factor, $\text{queryNorm}(q)$, enables a comparison between results of different queries; query coordination, $\text{coord}(q,d)$, gives higher scores to documents that contain a high percentage of terms in the query; query boosting, $t.\text{getBoost()}$, gives a boosted term more importance than another; and field length normalisation, $\text{norm}(t,d)$, gives higher weight to a shorter field than a long field in case a document is represented by more than one field, e.g. title and body.

In addition, during computation of the reduced query scores, we assign a specific query term boosting weight for each representation, $t.getBoost()$, equals the size of n -gram. The terms in q'_0 are not boosted, i.e., $t.getBoost() = 1$, since the original code tokens are 1-gram and can match relatively more frequently compared to other representations (we empirically validated this in Section 7.4.2). In contrast, the search terms in the n -gram-based representation q'_1, q'_2, q'_3 receive a higher query boosting weight. For example, if we choose the n -gram size for the query terms in q'_1 at 4, the matched n -grams in q'_1 will receive a boosting weight of 4. Since the query boosting score equals to the size of n -gram, the larger chosen n -gram size for each representation, the higher the query boosting weight is given and the higher score is received when terms in that representation find a match. We later explore that this boosting scores can be adjusted to accurately search for a specific clone type.

Finally, after the scores have been computed, the candidates are ranked based on their scores from the highest to the lowest. If two documents obtain the same score, they are sorted based on the alphabetical order of the file and method names. Siamese then returns the top n results from the ranked list to the user.

7.3.6 Incremental Updates

Siamese allows incremental updates to its index which is beneficial for maintaining an index of large-scale code repositories where the index can be updated to new changes without a need to reindex all the repositories again, similar to Hummel’s work [Hummel et al., 2010]. With large-scale source code data, it becomes a necessity for code clone detection or clone search tool to handle changes in code bases incrementally. Siamese leverages the flexibility of inverted index to allow its user to add, edit, delete code fragments in the index without affecting other indexed code fragments. For addition, the user can tell Siamese to incrementally add a given code fragment or project(s) to its index instead of recreating the index from scratch. For deletion, Siamese uses a given wildcard pattern for matching with the project or file name of code fragment(s) intended to be deleted and performs a deletion on the matched fragments. An update operation can be done using a deletion followed

by an addition.

7.4 Siamese Implementation

Our implementation of Siamese utilises *Elasticsearch* [Elasticsearch BV., 2016], an open-source high performance distributed full-text search engine, for a scalable code indexing and retrieval. The current implementation is in Java and uses a single Elasticsearch node with one shard. We built the preprocessing, MR module, QR module, and scoring function on top of Elasticsearch’s infrastructure. The Java method parsing is done using the Java parser [van Bruggen, 2017] and the tokenisation is done using the Antlr4 lexer with a Java 8 grammar [Parr et al., 2017]. Our implementation allows the tool to be executed on a single desktop machine or in a distributed manner by increasing the number of Elasticsearch nodes.

The MR, QR, scoring and ranking modules are language agnostic while the parser, tokeniser, and normaliser are language dependent. The current implementation of Siamese supports Java. To add a new language, one has to provide an implementation of the method extractor, tokeniser, and code normaliser for the language.

7.4.1 Selection of N -gram Sizes

The selection of the optimal n -gram size is not a trivial task. Selecting a large n -gram restricts Siamese to detect clones with small gaps of modified, inserted or deleted statements to ensure the confidence of being clones. In addition, a large n -gram size encodes more information in each gram and also contains a longer overlapping region between each gram, which will affect the memory required and the disk I/O time to process the n -grams. On the contrary, selecting a small n -gram allows larger gaps with better matching flexibility and requires less memory and disk access time, but also results in a higher chance of retrieving false clone pairs.

We surveyed the literature that use n -gram for clone detection and code similarity to study their choices of n -gram sizes. Burrows et al. [2007] selected 4-grams in their software plagiarism detection approach. Myles and Collberg [2005] found that the size of 4-gram or 5-gram offers a suitable tradeoff between credibility

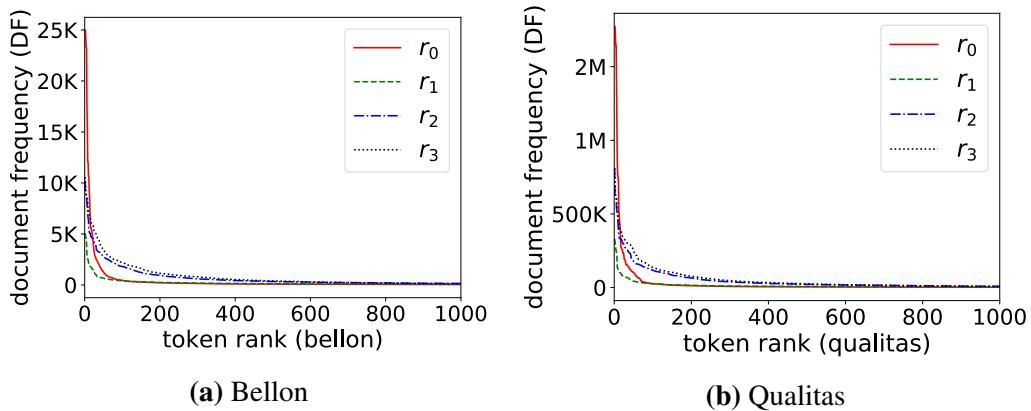


Figure 7.3: Java term rank and its document frequency

and resilience for their n -gram-based software birthmark technique. Ohmann and Rahal [2014] observed that $n = 4$ and $n = 13$ is the optimal choice for Manhattan and cosine distance respectively. We observed that 4-gram was chosen and shown a good performance in the three studies. Thus, we selected the n -gram size of 4 for our three code representation r_1 , r_2 , and r_3 in the MR module. 4-gram is long enough to capture code sequences but still allows small modifications within a statement. The representation r_0 relies on 1-gram to function as a keyword search, which is useful when looking for a specific token among the cloned fragments.

7.4.2 Choosing the Query Reduction Thresholds

Similar to the n -gram sizes, the selection of appropriate query reduction thresholds is important for generating high-quality queries. We used two data sets to select the optimal threshold θ values for the QR module. First, we selected the well-known Bellon’s clone benchmark [Bellon et al., 2007] for this analysis. The benchmark provides a partial clone ground truth in C and Java systems and has been used in several code clone studies [Wang et al., 2013b, Svajlenko and Roy, 2014, Koschke et al., 2006]. The Bellon’s benchmark was only used in this empirical n -gram analysis, and not used in any of Siamese’s evaluation to avoid configuration bias. We used the four Java systems, `java-swing` (204K SLOC), `eclipse-jdtcore` (148K SLOC), `eclipse-ant` (16K SLOC), and `netbeans-javadoc` (19K SLOC) from the benchmark. Second, we employed the Qualitas corpus [Tempero et al., 2010]. It is a curated Java corpus that has been used in several software engineering

studies [Taube-Schock et al., 2011, Beckman et al., 2011, Vasilescu et al., 2011, Omar et al., 2012]. The projects in the corpus represent various domains of software systems ranging from programming languages to visualisation. We selected the 20130901r release of the Qualitas corpus containing 111 Java open source projects. Since we need four threshold values for the four reduced query siblings q'_0 , q'_1 , q'_2 , and q'_3 , we derived four data sets from Bellon's benchmark and the Qualitas corpus, namely r_0 , r_1 , r_2 , and r_3 respectively, to match with the structure of our four code representations. For r_1 , r_2 , and r_3 , we adopted the n -gram sizes of 4 as previously discussed. Then, we counted document frequencies of the tokens and sorted them based on their frequency.

A visualisation of the term's document frequency vs. its rank from Bellon's benchmark is shown in Figure 7.3. We observed that the document frequency of r_0 , the original tokens, dropped sharply and started rapidly converging to one at approximately 10% (2K) of all the documents in the corpus. Similar observation was found for r_1 . The document frequency of r_2 and r_3 also converged to one. They dropped to one slightly slower than r_0 and r_1 due to the token normalisation, but they were also almost distinct at 10% of all the documents. Similar findings were observed for the Qualitas corpus as depicted in Figure 7.3b. With this observation, we picked the same query reduction threshold for all representations at 10%.

7.5 Experimental Design

We designed Siamese to be a multi-purpose clone search tool that can be exploited for various clone-related applications. To be useful, the tool must scale to the size of code corpora on the Internet while still return accurate ranked lists of clone results in a reasonable time (i.e., seconds).

We asked the following research questions to asses the practicality of Siamese to clone search applications.

RQ1: Multi-Representation and Query Reduction: *How effective are multi-representation and query reduction (MR-QR) to improve clone search accuracy?* To measure the effectiveness of our multi-representation and query

reduction techniques, we compared the search accuracy of Siamese with MR and QR reduction to the baseline text search engine.

RQ2: Search Accuracy: *How accurate are Siamese search results?* We measured Siamese search accuracy on the three established clone data sets and compared to state-of-the-art clone detection and clone search tools. The findings demonstrate the quality of Siamese’s search results compared to other tools.

RQ3: Clone Ranking: *How accurate is Siamese clone ranking?* We exploited Siamese MR for a fine-grained search targeting only Type-3 clones for alternative implementations and evaluated the accuracy of the ranked results.

RQ4: Scalability: *How practical is Siamese to index and search on large-scale code corpus?* Scalability is one of the most important aspects of Siamese. We evaluated Siamese’s scalability by measuring its indexing and querying time on the BigCloneBench data set containing 365M SLOC.

RQ5: Incremental Update: *How fast is Siamese’s incremental update?* Using an index of 130,719 GitHub projects, we evaluate Siamese’s incremental update module by measuring an index update time over hundreds of releases of the three most-starred Java software projects. The findings show the time saved by Siamese incremental index update when the user wants to update projects in the existing index.

7.5.1 Data Sets

We adopted three existing data sets used in other empirical code clone studies namely the OCD data set [Ragkhitwetsagul et al., 2018a], the SOCO data set [Flores et al., 2014], and the BigCloneBench data set [Svajlenko et al., 2014b, Svajlenko and Roy, 2016] for our evaluation. Their summary is displayed in Table 7.3. There is a complete ground truth for the first two data sets, while there is a partial ground truth for the third data set. The OCD data set provides Java files with pervasive code

Table 7.3: The data sets for Siamese evaluation

No.	Data set	Files	Clone pairs	SLOC
1.	OCD	100	10,000	9,618
2.	SOCO	259	453	26,122
3.	BigCloneBench	2,876,220	8,375,313	365M

modifications made by code obfuscators and compiler/decompilers. It covers clones of Type-1 to Type-4 (i.e., semantic clones or two code fragments with different syntax but share the same semantic). The OCD data set contains 100 Java files with a ground truth of 1,000 clone pairs at file-level. The 100 files consist of 10 groups of 10 files that are derived from one file and are therefore clones of each other. The SOCO data set was created for the detection of source code reuse competition. It contains clones of boiler-plate code fragments with a few or without modifications. The data set contains 259 files with a ground truth of 453 clone pairs at file-level. The OCD and the SOCO data sets were used in our previous study to compare 30 code similarity analysers [Ragkhitwetsagul et al., 2018a]. Third, the BigCloneBench data set is one of the largest clone benchmarks available to date. It is created from IJaDataset 2.0 [ASE group, 2018] of 25,000 Java systems. The benchmark contains 2.9 million files with 8 million manually validated clone pairs of Type-1 up to Type-4. The BigCloneBench data set was used for clone evaluation and scalability test in several large-scale clone detection and clone search studies [Kim et al., 2018, Li et al., 2017, Sajnani et al., 2016, Svajlenko et al., 2014b, Svajlenko and Roy, 2015]. Lastly, for the evaluation of Siamese incremental update, we relied on a set of publicly available 130,719 GitHub Java projects.

7.5.2 Error Measures

We evaluated our approach to answer RQ1 and RQ2 using three error measures: precision at 10, mean average precision (MAP), and mean reciprocal rank (MRR). They are defined as follows.

Given the top n ranked results of which TP are true positives, precision at n (denoted $\text{prec}@n$) is defined as

$$\text{prec}@n = \frac{\text{TP}}{n}. \quad (7.3)$$

Precision at 10 is a special case of precision at n where $n = 10$. It is used when only the top 10 results are taken into account, which reflects real-world web search scenarios that only 10 results are displayed per page [Manning et al., 2009].

Mean average precision (MAP) measures the quality of results across several recall levels where each relevant result is returned. It is calculated from multiple average precision (denoted APrec) obtained for the set of top k documents existing after each relevant document is retrieved, and this value is then averaged over all the queries [Manning et al., 2009]. If the set of relevant documents for a query $q_j \in Q$ is $\{d_1, \dots, d_{m_j}\}$ and R_{jk} is the set of ranked retrieval results from the top result until retrieving the document d_k , then

$$\text{MAP} = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{i=1}^{m_j} \text{APrec}(R_{jk}). \quad (7.4)$$

Mean reciprocal rank (MRR) considers the case where only one relevant document is needed. MRR measures, on average across $|Q|$ queries, the reciprocal rank of the relevant document (i.e., clone) to each query q in the search result [Craswell, 2009], i.e.,

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}. \quad (7.5)$$

7.6 Evaluation and Results

The evaluation of Siamese was performed on a single desktop computer. In RQ1, RQ2, and RQ3, Siamese was run on a MacBookPro with a single 2.9 GHz processor, 16 GB of RAM, and 512 GB of solid-state disk (SSD). In RQ4, Siamese was run on a CentOS 7.4.1708 machine with eight 3.00 GHz processors, 32 GB of RAM, and 500 GB SATA disk. In RQ5, Siamese was run on an Ubuntu 16.04.4 LTS machine with eight 3.70 GHz processors, 32 GB of RAM, and 2.8 TB SATA disk.

7.6.1 RQ1: Multi-Representation and Query Reduction

How effective are multi-representation and query reduction (MR-QR) to improve clone search accuracy?

To answer RQ1, we used the two data sets for which we knew the complete ground truth and measured the improvement of clone search offered by the multi-representation and query reduction (MR-QR) technique using MAP. To observe the clone search improvement offered by the MR module; the QR module; and the combination of MR-QR to a traditional search engine, we compared the baseline text search engine represented by Elasticsearch to three variants of Siamese: (1) Siamese with MR, (2) Siamese with QR, and (3) Siamese with MR-QR. The baseline represents code search engines that rely on keyword search of source code fragments, and do not take code structure into account. Moreover, the baseline of Elasticsearch text search engine is adopted by GitHub to search for code in its 8 million code repositories³. Thus, the baseline also represents the code search capabilities of GitHub. For the OCD data set, we retrieved 100 queries from the ground truth and expected 10 clones at the top for each query result ($r = 10$). For the SOCO data set, the 453 clone pairs in the ground truth came from 115 unique files, which we used as the queries. The number of relevant results r for each query was varied and based on the number of cloned files associated with each query as specified in the ground truth.

We started by evaluating the clone search performance based on 15 unique combinations of code representations as displayed in Table 7.4, denoted by the subscripted number. For example, r_{123} represents the combination of r_1 , r_2 , and r_3 . For the OCD data set, Siamese already performed decently well using one representation especially r_1 with the MAP of 0.921. The highest MAP score was from a combination of r_{13} at 0.938. The combination of four representations (r_{0123}) received a slightly lower MAP of 0.900. However, we will show later that by using query reduction to get rid of the extraneous tokens, we could obtain even higher

³<https://www.elastic.co/use-cases/github>

MAP scores than using r_1 and r_{13} . For the SOCO data set, the best combination was using a single representation of r_1 at 0.990. The r_1 representation performed well with the SOCO data set because it contained clones of boiler-plate code with very few changes (Type-1 clones). Thus, using the n -gram sequences of original tokens in r_1 would match best with the clones. The combination of four representations gave the MAP of 0.976, slightly lower than r_1 .

The results from Table 7.4 shows that there is no single representation that performs well on both data sets. We could sacrifice some level of search precision by combining code representations to be able to locate different types of clones in different data sets without changing the configurations. This supports our intuition of using multiple code representation for clone search.

By adopting the multi-representation alone, we could gain a higher MAP score than the baseline by up to 15% (from 0.785 to 0.900). By applying QR on top of the baseline text search, we also received an improvement. As displayed in Table 7.5, The MAP score on OCD increased by about 18% (from 0.785 to 0.926). However, we observed a slight decrease of MAP after applying QR for the SOCO data set with the MAP score decreased from 0.977 to 0.975. Thus, using QR alone can be beneficial only in some cases. Nevertheless, after combining MR and QR together, we always obtained the highest MAP for both two data sets. The MAP scores increased to 0.953 for OCD and to 0.991 for SOCO.

To confirm our findings of improvements by MR-QR, we performed a statistical test using a two-tailed non-parametric randomisation test due to its robustness in information retrieval [Smucker et al., 2007]. Our null hypothesis (H_0) was that there is no significant difference between the results from the baseline to the results of Siamese using MR-QR. We performed the test using 100,000 random samples with a confidence interval value of 99% (i.e., $\alpha \leq 0.01$). The values in bold represent a statistically significant improvement which rejects the null hypothesis. We found that MR-QR helps to improve the clone search precision with statistical significance compared to the baseline on the OCD data set. The improvement for SOCO was not statistically significant due to an already high MAP score reported by the baseline.

Table 7.4: Search performance (MAP) with different combinations of code representations

(a) One and two representations

Data set	1 rep.				2 reps.					
	r_0	r_1	r_2	r_3	r_{01}	r_{02}	r_{03}	r_{12}	r_{13}	r_{23}
OCD	0.785*	0.921	0.889	0.892	0.850	0.844	0.842	0.923	0.938	0.900
SOCO	0.977*	0.990	0.948	0.939	0.987	0.964	0.960	0.979	0.978	0.942

* = using the same representation as the baseline (token-based keyword search)

(b) Three and four representations

Data set	3 reps.				4 reps.
	r_{012}	r_{013}	r_{023}	r_{123}	r_{0123}
OCD	0.885	0.882	0.865	0.930	0.900
SOCO	0.979	0.978	0.956	0.971	0.976

Table 7.5: Search performance improvement (MAP) after adding multi-representation and query reduction

Data	Settings				p -value	A_{12}
	Baseline	MR	QR	MR-QR		
OCD	0.785	0.900	0.926	0.953	0.000	0.743
SOCO	0.977	0.976	0.975	0.991	0.205	0.522

We complemented the statistical test by employing a non-parametric effect size measure called Vargha and Delaney's A_{12} measure [Vargha and Delaney, 2000] to measure the level of differences between two populations and found that the effect size on the OCD data set is large (0.743), while on the SOCO data set is negligible (0.522). These findings show that MR-QR is highly effective against clones with several modifications applied to the source code, and mildly effective against clones with boiler-plate code or exact code copies.

To answer RQ1, the adoption of MR-QR improves the clone search performance compared to the baseline text search engine with statistical significance. The inclusion of MR and QR alone increases the search accuracy in most of the cases.

7.6.2 RQ2: Search Accuracy

How accurate are Siamese search results?

We utilised all three data sets to measure Siamese's search precision. Each of them is discussed separately below.

7.6.2.1 OCD and SOCO

To answer RQ2, we utilised all three data sets to measure Siamese's search precision. For the OCD and SOCO data set, we compared Siamese using MAP to seven state-of-the-art clone detectors at file level. The other clone detectors included SourcererCC [Sajnani et al., 2016], CCFinderX [Kamiya et al., 2002], DECKARD [Jiang et al., 2007a], iClones [Göde and Koschke, 2009], JPlag [Prechelt et al., 2002], NiCad [Roy and Cordy, 2008], and Simian [Harris, 2003]. For CCFinderX, DECKARD, iClones, JPlag, NiCad, and Simian, we relied on the results reported in Chapter 5. For SourcererCC, we followed the method shown in Chapter 5 to automatically compute a similarity score based on the clone pairs reported by the tools, create a ranked results based on the similarity scores, and measure MAP score. We also tuned Siamese and compared the optimised Siamese to the other tools' optimised configurations as a previous study by Wang et al. [2013b] and our results in Chapter 5 have shown that the default configurations of clone detectors could harm the validity of studies relying on them.

The mean average precision of Siamese compared to seven other clone detectors using their default configurations on the two data sets, OCD and SOCO, is displayed in Table 7.6 and Table 7.7. With the default configurations of n -gram sizes and query reduction thresholds (θ) derived from the empirical study, Siamese performed best on the OCD data set with MAP of 0.953 and for the SOCO data set, Siamese was ranked first along with JPlag with MAP of 0.991.

Regarding the optimised version, we tuned Siamese's n -gram sizes and θ to give the highest MAP score. The n -gram sizes for the three code representation r_1 , r_2 , and r_3 starts from 4 to 24 with an increasing step of 4 (the representation r_0 always has the n -gram size of 1). We tried the four n -gram sizes on the three

representation independently and obtained 216 different combinations. The query reduction thresholds θ cover 2%, 4%, 6%, 8%, and 10% and were set identically for the four representations. Combined the two parameters together, we searched for 1,080 combinations of Siamese's configurations. The other tools' optimised configurations and their parameter search space are based on the results from Chapter 5. CCFinderX and JPlag was ranked 1st for OCD and SOCO with MAP of 1.000 and 0.997 respectively. Although Siamese did not give the highest MAP scores based on the optimised configurations, it still offered a very high MAP score (0.997 and 0.994) and was ranked the 2nd for both OCD and SOCO. Moreover, Siamese always outperformed SourcererCC, DECKARD, iClones, NiCad, and Simian in both the default and the optimised configurations. Although it gave slightly lower MAP score than CCFinderX and JPlag after tuning, Siamese offered a much higher scalability than the two clone detectors as will be shown in RQ4.

The multi-representation module helped Siamese to perform well on different data sets even without tuning as we observed that the optimised MAP values were very close to the default configurations' MAP values. In practice, it is very difficult to always tune a clone detector to their optimal performance. We could optimise the clone detectors in this study because we knew the complete clone ground truth of the OCD and the SOCO data sets as they were generated data sets. A clone ground truth does not exist in software projects. Thus, we mostly have to rely on the default configuration of the clone detection tools. Moreover, Chapter 5 shows that although we could find the tools' optimal configurations on one data set, we cannot efficiently reuse them on another data set. The results in Table 7.6 and Table 7.7 suggest that Siamese's search performance, with or without tuning, was still comparable or even better than other tools with their optimised configurations. This shows that Siamese effectively handles clones with several code modifications in the OCD data set and boiler-plate code in the SOCO data set, while still offers comparable search precision to other clone detection tools optimised for the data sets.

Table 7.6: Comparison of search performance (MAP) on the OCD data set (100 queries)

Tool	Default		Optimised	
	Settings	MAP	Settings	MAP
Siamese	$r_1=4\text{-gram}$, $r_2=4\text{-gram}$, $r_3=4\text{-gram}$, $\theta=10\%, 10\%, 10\%$	0.953	$r_1=[4,8,12,16,20,24]\text{-gram}$, $r_2=24\text{-gram}$, $r_3=8\text{-gram}$, $\theta=2\%, 2\%, 2\%, 2\%$	0.997
Text Search	N/A	0.785	–	–
SourcererCC	similarity=80%	0.471	similarity=40%	0.848
CCFinderX	b=50, t=12	0.569	b=5, t=11	1.000
DECKARD	mintoken=50, stride=inf similarity=1.0	0.665	mintoken=30, stride=1 similarity=0.95	0.924
iClones	minblock=20, minclone=100	0.444	minblock=10, minclone=50	0.668
JPlag	t=9	0.857	t=5	0.918
NiCad	UPI=0.30, minline=10, rename=none, abstract=none	0.457	UPI=0.50, minline=10, rename=blind, abstract=declaration	0.859
Simian	threshold=6	0.442	threshold=3, ignoreIdentifiers	0.916

Table 7.7: Comparison of search performance (MAP) on the SOCO data set (115 queries)

Tool	Default		Optimised	
	Settings	MAP	Settings	MAP
Siamese	$r_1=4\text{-gram}$, $r_2=4\text{-gram}$, $r_3=4\text{-gram}$, $\theta=10\%, 10\%, 10\%$	0.991	$r_0=1\text{-gram}$, $r_1=[4,8,12,16,20,24]\text{-gram}$, $r_2=4\text{-gram}$, $r_3=16\text{-gram}$, $\theta=8\%, 8\%, 8\%, 8\%$	0.994
Text Search	N/A	0.977	–	–
SourcererCC	similarity=80%	0.776	similarity=60%	0.839
CCFinderX	b=50, t=12	0.942	b=5, t=9	0.982
DECKARD	mintoken=50, stride=inf similarity=1.0	0.946	mintoken=30, stride=2 similarity=0.95	0.980
iClones	minblock=20, minclone=100	0.799	minblock=8, minclone=70	0.882
JPlag	t=9	0.991	t=8	0.997
NiCad	UPI=0.30, minline=10, rename=none, abstract=none	0.870	UPI=0.30, minline=5, rename=blind, abstract=literal	0.931
Simian	threshold=6	0.884	threshold=4, ignoreIdentifiers	0.978

7.6.2.2 BigCloneBench

The third data set is the BigCloneBench, which is a well-known data set that has been used to benchmark code clone detectors and clone search engines [Sajnani et al., 2016, Kim et al., 2018, Li et al., 2017]. Because Siamese is not a clone detector but a clone search tool, it does not report a set of clones that can be used to measure recall and precision. Nevertheless, we compared its performance to other tools here for a situation where it will be adapted as a clone detector.

The BigCloneBench data set's size represents code corpora on the Internet and is suitable to assess how well the tool differentiates and reports relevant code snippets from millions of candidates. Moreover, the data set offers a ground truth of 8 million clone pairs. The evaluation was performed at method level as required by the BigCloneBench oracle. We measured Siamese on both clone recall and precision. Both evaluations are done by issuing multiple queries and evaluated the returned ranked results.

Recall: We followed the approach used by Kim et al. [2018], who also evaluated their clone search engine for recall, by choosing 14,780 methods that appeared in the clone oracle as the queries. Although we did not use all the methods in BigCloneBench to query (similar to Kim et al. [2018]), it does not affect the clone recall. The methods that do not appear in the clone oracle do not have any clone pair associated with them, thus using them to query for clones would only result in false positives, which is not taken into account for recall (on the contrast, it will affect precision). To compute the recall score, we utilised an automated tool called BigCloneEval [Svajlenko and Roy, 2016] which was created for recall computation on BigCloneBench. For each query, we choose the result size of 900 to match with the settings used in the evaluation of a clone search engine, FaCoy, by Kim et al. [2018]⁴. After finishing querying, we gave the result to the BigCloneEval tool for recall calculation. Table 7.8, Table 7.9, and Table 7.10 shows the recall scores of Siamese on BigCloneBench compared to the other five tools including SourcererCC, CCFinderX, DECKARD, iClones, and NiCad as reported in the study

⁴Due to the release of the FaCoy tool as only a virtual machine image, we could not include it in our other RQs.

by Sajnani et al. [2016] and FaCoy code search tool as reported in the study by Kim et al. [2018]. BigCloneBench categorised the clone pairs into Type-1 (T1), Type-2 (T2), very-strongly Type-3 (VST3) with a similarity in range of 90% (inclusive) to 100%, strongly Type-3 (ST3): 70–90%, moderately Type-3 (MT3): 50–70%, and weakly Type-3 or Type-4 (WT3/T4): 0–50% [Svajlenko and Roy, 2016]. Moreover, BigCloneEval divides the evaluation into 3 sets: All Clones, Intra-Project Clones, and Inter-project Clones. We included the other tools’ results for the all three sets except the FaCoy tool which reported its recall scores only for the All Clones set.

For All Clones set (Table 7.8), Siamese provided recall scores of 99% for T1, T2, VST3, and ST3. Siamese obtained the highest recall of 88% for MT3 compared to other tools and 17% for WT3/T4. When dividing into Intra-Project (Table 7.9) and Inter-Project clones (Table 7.10), Siamese performed slightly better on both sets with higher or the same recall scores as in the All Clones set. Interestingly, we found that Siamese could return 99% of MT3 clones in Intra-Project clones while other tools reported up to 14%. Similarly, Siamese returned 77% of WT3/T4 clones while CCFinderX and DECKARD reported only 1% of the clones. A similar finding was observed for Inter-Project clones where Siamese obtained the highest recall at 87% of MT3 clones and 16% of WT3/T4 clones. The results show that the multi-representation and query reduction techniques enable Siamese to find more challenging clone pairs than state-of-the-art techniques. Although Siamese and SourcererCC share fundamental concept of index-based and token-based clone detection, Siamese can offer higher clone recall for the challenging clone types of ST3, MT3, and WT3/T4 because it does not remove any token from the code in the index. On the other hand, SourcererCC’s partial indexing keeps only rare tokens in the clone index, which restricts the tool to find only clones that share the rare tokens. The removal of frequent tokens in Siamese occurs at a query time and it only affects the tokens in the query. However, Siamese suffers from a larger clone index than SourcererCC due to the complete collection of code tokens and also its multiple code representations.

Precision: To measure precision, there is no benchmark and standard method-

Table 7.8: BigCloneBench Recall Measurements (All Clones)

Tool	All Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4
<i>Clone search engines</i>						
Siamese	99	99	99	99	88	17
FaCoy [Kim et al., 2018]	65	90	67	69	41	10
<i>Clone detectors [Sajnani et al., 2016]</i>						
SourcererCC	100	98	93	61	5	0
CCFinderX	100	93	62	15	1	0
DECKARD	60	58	62	31	12	1
iClones	100	82	82	24	0	0
NiCad	100	100	100	95	1	0

Table 7.9: BigCloneBench Recall Measurements (Intra-Project Clones)

Tool	Intra-Project Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4
<i>Clone search engines</i>						
Siamese	100	99	100	100	99	77
FaCoy [Kim et al., 2018]	—	—	—	—	—	—
<i>Clone detectors [Sajnani et al., 2016]</i>						
SourcererCC	100	99	99	86	14	0
CCFinderX	100	89	70	10	4	1
DECKARD	59	60	76	31	12	1
iClones	100	57	84	33	2	0
NiCad	100	100	100	99	6	0

Table 7.10: BigCloneBench Recall Measurements (Inter-Project Clones)

Tool	Inter-Project Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4
<i>Clone search engines</i>						
Siamese	99	100	99	99	87	16
FaCoy [Kim et al., 2018]	—	—	—	—	—	—
<i>Clone detectors [Sajnani et al., 2016]</i>						
SourcererCC	100	97	86	48	5	0
CCFinderX	98	94	53	1	1	0
DECKARD	64	58	46	30	12	1
iClones	100	86	78	20	0	0
NiCad	100	100	100	93	1	0

Table 7.11: BigCloneBench Precision Measurements

MRR	prec@10	T1	T2	T3
0.948	0.871	0	25	811

ology for precision measurement in clone detection and some authors relied on a manual investigation of the reported clone pairs [Sajnani et al., 2016]. The BigCloneBench is created by using regular expressions derived from 43 target functionalities, i.e., Java class files, to search for clone candidates in 25,000 Java projects, followed by a manual confirmation. Thus, we chose the 43 Java files that represented the target functionalities in BigCloneBench as the search queries. We obtained 96 method queries from the 43 chosen files. The oracle of 8 million manually validated clone pairs provided by the BigCloneBench authors is only a partial ground truth as it only contains validated clone pairs but not all existing clone pairs. It is possible that Siamese reports true clones which do not exist in the ground truth during the evaluation. Thus, a manual validation is needed to obtain precision scores. To evaluate Siamese as a clone search engine that returns a ranked list of top n results, we relied on MRR and precision at 10 for precision measurement. The two error measures are well-known in information retrieval since they reflect a real-world setting of using a search engine where only a few first results will be looked at due to a limited attention span of human investigator [Miller, 1956]. The first author took the role of a human investigator.

Table 7.11 shows the MRR and precision at 10 scores based on the ground truth in the benchmark and after the manual confirmation. Siamese’s search results of the 96 queries on BigCloneBench offered an MRR score of 0.948 and precision at 10 of 0.871. The MRR score of 0.948 shows that Siamese mostly returns true clone fragment as the first result. The precision at 10 score of 0.871 shows that true clones are observed within the top ten on average 87.1% of the time. These are relatively high precision scores considering that there was no Type-1 clone for all the 96 queries and only Type-2 and Type-3 clones were available.

During our manual confirmation of the BigCloneBench clone search results,

we noticed some interesting clones that were reported by Siamese. We found that in a few cases, Siamese not only reported clones that were syntactically similar to the query but also semantically similar. For example, consider the `binarySearch1` method shown before in Section 7.3.3 as the query. As shown in Figure 7.4, the first result was very similar to the query but with differences in the data types and expressions. The 2nd result contained a more diverse version of binary search with renamed variables and different conditional statements. Interestingly, we found that the 3rd result is a method that performed binary search using a `while` loop instead of recursion as in the query and the 5th result was a method to search for an index number which used binary search as the underlying search algorithm.

7.6.2.3 False Positives

To understand the weaknesses of our approach, we summarise a few patterns found in the manually-validated false positive clone pairs. First, a number of false positive clone pairs come from a method that is declared inside another method. For example, as shown in Figure 7.5, the method `deleteRecursively1` is reported as clone pairs with its three inner methods: `visitFile`, `visitFileFailed`, `postVisitDirectory`. This problem can be fixed by analysing the clone results and filtering these inner-method clone pairs out.

In addition, we observed that many of the false positive pairs are caused by two methods that share several code tokens and n -gram sequences. As shown in Figure 7.6, the two methods perform a different task of checking a palindrome word and checking for blank string. Nonetheless, they share several similar code tokens such as `for (int i = length - 1; i >= 0; i--)`, `.charAt(i)`, or `length = original.length();`. Increasing the n -gram sizes may remove these false positives, while also reduce the chance of finding Type-3/Type-4 clones.

To answer RQ2, Siamese offers the highest mean average precision on two clone benchmarks compared to seven clone detectors. Its multi-representation enables Siamese to detect challenging Type-3 and Type-4 clone pairs better than other tools, while still reserves high recall on Type-1, and Type-2 clones. It offers the highest recall scores of ST3, MT3, and WT3/T4 clone pairs on BigCloneBench.

```

/* 1st result (sample/BinarySearch.java, 19, 30) */
public static <T extends Comparable<T>>
    int binarySearch3(T[] arr, T key, int imin, int imax) {
    if (imax < imin) return -1;
    int imid = (imin+imax)/2;
    if (arr[imid].compareTo(key) > 0)
        return binarySearch3(arr, key, imin, imid-1);
    else if (arr[imid].compareTo(key) < 0)
        return binarySearch3(arr, key, imid+1, imax);
    else return imid;
}

/* 2nd result (default/103246.java, 20, 26) */
private int recFind(long searchKey,
                    int lowerBound, int upperBound) {
    int curIn;
    curIn = (lowerBound + upperBound) / 2;
    if (a[curIn] == searchKey) return curIn;
    else if (lowerBound > upperBound)
        return nElems;
    else {
        if (a[curIn] < searchKey)
            return recFind(searchKey, curIn + 1, upperBound);
        else
            return recFind(searchKey, lowerBound, curIn - 1);
    }
}

/* 3rd result (selected/2663331.java, 292, 299) */
double getValueForFeature(int f) {
    int imin = 0, imax = features.length;
    while (imin < imax) {
        int mid = (imin + imax) / 2;
        if (features[mid] == f) return values[mid];
        else if (features[mid] > f) imax = mid;
        else imin = mid + 1;
    }
    return 0;
}

/* 5th result (selected/541979.java, 138, 144) */
private int getIndex(int c, int start, int stop) {
    int pivot = (start + stop) / 2;
    if (c == value[pivot]) return pivot;
    if (start >= stop) return -1;
    if (c < value[pivot]) return getIndex(c, start, pivot - 1);
    return getIndex(c, pivot + 1, stop);
}

```

Figure 7.4: Search results with syntactic and semantic clones

```

public static void deleteRecursively1(Path dir) throws
    IOException {
    Files.walkFileTree(dir, new SimpleFileVisitor<Path>() {
        @Override
        public FileVisitResult visitFile(Path file,
            BasicFileAttributes attrs) throws IOException {
            Files.delete(file);
            return FileVisitResult.CONTINUE;
        }

        @Override
        public FileVisitResult visitFileFailed(Path file, IOException
            exc)
            throws IOException {
            Files.delete(file);
            return FileVisitResult.CONTINUE;
        }

        @Override
        public FileVisitResult postVisitDirectory(Path dir,
            IOException exc)
            throws IOException {
            if (exc == null) {
                Files.delete(dir);
                return FileVisitResult.CONTINUE;
            } else {
                throw exc;
            }
        }
    });
}

```

Figure 7.5: A false positive clone pair containing methods inside a method

7.6.3 RQ3: Clone Ranking

How accurate is Siamese clone ranking?

In this RQ, we evaluated Siamese clone ranking to report alternative implementations, i.e., Type-3 or Type-4 clones, on the top of the search results. This clone ranking is useful for finding bug fix candidates or similar implementations. Although RQ2 shows that Siamese returns the largest number of Type-3 and Type-4 clones from BigCloneBench, the recall evaluation did not take the ranking into accounts. With the multiple code representations, we are allowed to search for a specific type of clones that fits our needs. By adjusting a different boosting score for each representation at a query time, we could target clones of a specific type

```

/* QUERY - TestPalindrome.java, 2, 13 */
public static boolean isPalindrome(String original) {
    //A not very efficient example
    String reverse = "";
    int length = original.length();
    for (int i = length - 1; i >= 0; i--)
        reverse = reverse + original.charAt(i);
    if (original.equals(reverse))
        return true;
    else
        return false;
}

/* 1st RESULT - 2394080.java, 118, 125 */
public static boolean isNotBlank(String str) {
    int length;
    if ((str == null) || ((length = str.length()) == 0)) return
        false;
    for (int i = length - 1; i >= 0; i--) {
        if (!isWhitespace(str.charAt(i))) return true;
    }
    return false;
}

```

Figure 7.6: Another false positive clone pair containing similar code tokens

to be ranked on top of the search results, while discriminating against clones of unwanted types to be ranked lower.

This clone ranking is difficult or impossible to achieve by traditional clone detection tools. First, tools like CCFinderX, NiCad, or SourcererCC do not provide a ranked list of clones. So a human investigator does not know which clone pairs to start the investigation and has to rely on random sampling. Second, although we can rank the clone pairs based on their similarity score (CCFinderX and NiCad can report similarity scores), we cannot explicitly select clones of a specific type to be on top of the list. For example, let say we are searching for alternative implementations of a buggy code fragment, and we use NiCad for this task. We do not want Type-1 or Type-2 clones because they are identical or very similar to the buggy code fragment that we currently have. Thus, we configure NiCad to find Type-3 clones. Nonetheless, since Type-3 clones subsume Type-2 and Type-1 clones by definition, we cannot use NiCad to locate *only* Type-3 clones. The Type-1 clones reported by NiCad will always have a similarity score higher than Type-2 and Type-3 clones,

Table 7.12: Type-3-only search: the boosting scores for the four code representations and the search accuracy

Tool	Representations				MRR
	r_0	r_1	r_2	r_3	
Baseline (text search)	1	–	–	–	0.4633
Siamese (default)	1	4	4	4	0.4550
Siamese (boosted)	10	-1	-1	1	0.7050

and will always be ranked on top. The human investigator will have to manually go through a number of Type-1 and Type-2 clones before finding the Type-3 clones that he or she is looking for. Third, most of the clone detectors locate clones based on a given similarity threshold. SourcererCC’s partial indexing only keeps code tokens that form a clone pair with similarity higher than or equal to the threshold. Since this decision is made at indexing time, a change of the similarity threshold to find stricter or more relaxed clones will result in re-indexing of the code base. Other clone detectors such as Simian, DECKARD, or iClones would face the same issues.

Similar to RQ2, we used the BigCloneBench index with 8.1 million code fragments and performed the clone search based on the 96 queries which represented 43 target functionalities in BigCloneBench. They were chosen again for this RQ because the 96 queries contained general functionalities that were normally found in Java programs, such as binary search, bubble sort, file copy, and extraction of a compressed file. Moreover, the benchmark’s partial clone ground truth helped us in the manual clone investigation step. The maximum number of clone results to be investigated is 10.

Since the 43 target functionalities had only Type-2 and Type-3 clone pairs and did not have any Type-1 clone pair in BigCloneBench, we injected them into the index so they could appear in the search results. We intentionally added them into the search index in order to confuse the tool. Our goal was to find Type-3 clones that slightly or moderately differ from the query, so the injected Type-1 clones should not appear on top of the search results.

The adjusted boosting scores of Siamese for Type-3 clone search is shown in Table 7.12. The original and Type-3 representations r_0 and r_3 received positive boosting scores of 10 and 1 respectively, while r_1 and r_2 received negative boosting scores of -1. This setting was suitable for finding clones that deviate from the query because the literal clones (Type-1) and parameterised clones (Type-2) were penalised with the negative boosting scores. We need to keep positive boosting scores for tokens in the original representation r_0 to get rid of false positives due to accidental structural similarity matches on r_3 . We gave a higher score of ten for r_0 than one of r_3 to push Type-3 clones with similarity keywords on the top of the list. Since there was no clone detection in our study that gives ranked list of clones, we compared Siamese to the baseline text search engine, i.e., using the source code original tokens with no boosting score (boosting score equals one), and Siamese with the default configurations with the boosting scores of 1 for r_0 and 4 for r_1 , r_2 , and r_3 .

We adopted MRR to measure the search accuracy⁵. Since the goal of this RQ is to find bug fix candidates or alternative implementations, we only targeted Type-3 clones. Clones with Type-1 or Type-2 were not considered as relevant and received a zero score when computing MRR. Thus, in this case, the MRR score reflected how well the tool retrieved Type-3 clones on the top of the search results. We consulted the BigCloneBench clone oracle to validate the returned clone pairs and their clone types. When a clone pair could not be found in the clone oracle, the thesis author performed a manual validation of the clones.

The MRR scores of the baseline text search and Siamese are displayed in Table 7.12. The baseline always returned Type-1 clone pairs on top of the search results (96 times out of 96 queries), followed by Type-2 and Type-3 clones and received an MRR score of 0.4633. The default Siamese gave a similar performance with an MRR score of 0.4550. Boosted Siamese outperformed the other two tools with a higher MRR score of 0.7050. The boosted Siamese returned Type-3 clone

⁵We were deterred from using the well-known Normalised Discounted Cumulative Gain (NDCG) that was suitable for assessing the quality of ranked results. NDCG needs a complete ground truth of relevant documents which were not the case for BigCloneBench.

```

/* QUERY - Fibonacci.java, 3, 10 */
public static int getFibonacci(int n) {
    if(n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return getFibonacci(n-1) + getFibonacci(n-2);
}

/* 1st RESULT - 2156644.java, 5, 13 */
public int getFibonacci(int n) {
    int prev[] = { 1, 1 };
    for (int i = 1; i < n; i++) {
        int aux = prev[1] + prev[0];
        prev[0] = prev[1];
        prev[1] = aux;
    }
    return prev[1];
}

```

Figure 7.7: An example of Type-3/Type-4 cloned fragment returned as the 1st result

pairs on the top result 59 times, returned Type-1 clone pairs on the top 23 times, and did not return any correct clone pairs 14 times. Figure 7.7 shows an example of a query and a Type-3 clone fragment returned by Siamese. The pair are both methods to get a Fibonacci number. They share the same input/output but contain two different implementations using recursion and a for loop.

To answer RQ3, Siamese can effectively search and return a specific type of clones on top of the search results. Due to its multi-representation of code, Siamese can target which type of clones to be ranked on the top while at the same time discriminates clones of unwanted types. This specific clone-type ranking cannot be done using existing clone detection tools or code search due to its use of a single code representation. The search is beneficial for a case where only a specific type of similar code is needed, such as finding potential bug fix candidates which are not identical to the given query.

7.6.4 RQ4: Scalability

How practical is Siamese to index and search on large-scale code corpus?

We evaluated Siamese’s scalability by measuring the time needed to index and query various code base sizes. We created 10 sets of Java code with different sizes by randomly selecting files from BigCloneBench. The number of files in a set i , $i \in [1, 10]$, is 2^{2i} . The smallest set has 4 files (22 methods) and the largest set has 1,048,576 files (1,771,183) methods. We also added the complete BigCloneBench data set with 2.9 million files (4,870,113 methods) as the last (11th) set. The experiment was performed on a CentOS 7.4.1708 machine with eight 3.00 GHz processors, 32 GB of RAM, and 500 GB SATA disk.

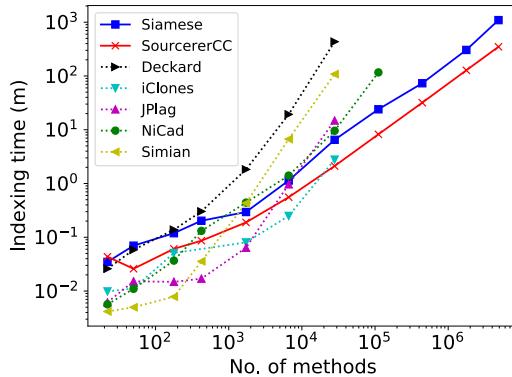
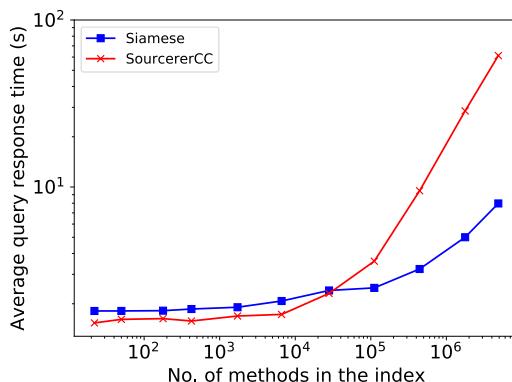
We separately measured the tools’ index and query time in this RQ because we are more interested in a scenario of clone search than clone detection. In the clone detection scenario as performed by Koschke [2014] or SourcererCC [Sajnani et al., 2016], it is a one-off process. An index of code bases is created. Then, queries containing code fragments either from within the same project (intra-clone detection) or from other projects (inter-clone detection) are issued on the index to locate clones. The clone index may be kept for later uses if needed or recreated if the analysed code bases change. In this scenario of one-off clone detection, indexing and querying occur one after the other in a single execution. On the other hand, in the clone search scenario (or incremental and real-time detection as presented by Hummel et al. [2010]), an index of large code bases is persisted only once and loaded into memory whenever the clone search engine starts. The index is frequently updated to reflect the changes in the code bases. With this approach, a clone search tool allows as many queries as needed without a need to reindex the code bases again. We can tolerate slow indexing time as long as the tool offers fast querying time, which occurs much more often. Thus, measuring both the index and query time allows us to know how long it takes to prepare the index, and how long it takes to only retrieve clones.

For the indexing phase, we compared our tool to seven clone detectors: SourcererCC, CCFinderX, DECKARD, iClones, JPlag, NiCad, and Simian. Since all the tools except Siamese and SourcererCC do not separate their clone detection into indexing and querying phase, we use their main command to detect clones to

execute. Moreover, the other five clone detectors besides SourcererCC do not use an indexed-based approach, so we cannot directly compare their indexing time and rely on their clone detection time as the indexing time. We included them in this comparison to assess their scalability to large-scale code data. For Siamese and SourcererCC, we specifically instructed the tools to perform indexing on the given data sets. Each tool was executed using their default configurations and, if allowed by the tool, we allocated the same amount of 8GB of memory for their execution. We measured the execution time using the `time` command. Unfortunately, during the execution of CCFinderX, the tool reported encoding errors on several files. We needed to remove those files from the data set to run the tool, which would affect its running time. So, we decided to remove CCFinderX from this evaluation.

The tools’ indexing time is displayed in Figure 7.8. The plot shows the tools’ execution time against the number of methods in each data set. Every tool completed their analysis of 22, 50, 178, 423, 1723, 6.6K, and 28K methods with increasing execution time. DECKARD reported clones in the 28K set in 7 hours 14 minutes and did not return any result on the 111K set within a week, so we decided to terminate the tool’s execution. iClones and JPlag finished their executions on the 6.6K set in 3 minutes and 15 minutes respectively and ran out of memory on the 28K set. NiCad threw an error in cross-clone analysis on the 442K-method set. Simian reported clones in the 28K set within 1 hour and 48 minutes and failed to analyse the 442K set.

Siamese and SourcererCC were the only two tools that could complete their indexing of the 11 data sets. SourcererCC finished indexing 111K, 442K, and 1.7M methods within 8 minutes, 32 minutes, and 2 hours respectively. For the complete BigCloneBench (4.8M methods, 365M SLOC), SourcererCC used 6 hours to complete the indexing. Siamese finished indexing the same data sets within 24 minutes, 1 hour 13 minutes, and 5 hours respectively. For the complete BigCloneBench, Siamese took 18 hours 13 minutes to finish the indexing. Since Siamese derives multiple code representations from given code fragments, its indexing time took around three times longer than SourcererCC.

**Figure 7.8:** Indexing time (minutes)**Figure 7.9:** Querying time (seconds)

For the querying phase, we compared Siamese only to SourcererCC because it is the only tool besides Siamese that successfully scaled to the full BigCloneBench data set. Moreover, it also works in a two-phase approach of indexing and querying like Siamese. Both Siamese and SourcererCC were configured with their default configurations, and only methods with at least ten lines were considered. After each subset was indexed into Siamese's and SourcererCC's index, we performed 100 queries and measured the query response time. We used a fixed set of 100 randomly selected files from BigCloneBench as the queries. One query was issued at a time and the average query time was derived from the execution time of all the queries as shown in Figure 7.9. We observed a sharply increasing query response time from SourcererCC when the number of methods in the index grew. Since SourcererCC is designed for detecting clones within a data set, it has its optimal speed when a large collection of files is given as an input and is processed in a

batch. Nevertheless, SourcererCC does not respond fast when it comes to a single query because it has to load the index into memory every time a query is issued. On 111K; 442K; and 1.8M methods in the index, SourcererCC’s query took 3.4, 9.3, and 28.3 seconds on average. Siamese offered a slightly increasing query response time of 2.5, 3.2, and 5.0 seconds on the same sets of 111K; 442K; and 1.8M methods. On the full BigCloneBench data set with 4.9M methods, Siamese’s query time increased slightly to 8 seconds while it took SourcererCC 60 seconds to return the results. This shows that Siamese is suitable for situations where one query is issued at a time, such as searching for code examples, finding similar code candidates for program repairs, or checking for cloned code from the Internet.

To answer RQ4, Siamese offers higher scalability than traditional clone detectors including DECKARD, iClones, JPlag, NiCad, and Simian. It scales to a large code corpus of 4.9M methods with 365M SLOC in less than a day. Its indexing time is slower than SourcererCC, but it offers a faster query response time within 8 seconds. Siamese’s query response time is marginally affected by the index size. We observed 3 seconds increment in the query time even when the index size grew three times larger.

7.6.5 RQ5: Incremental Updates

How fast is Siamese’s incremental update?

We followed the same approach used by Hummel et al. [2010] to evaluate the incremental update capability of Siamese. We instructed Siamese’s to update versions of software projects in an index of 130,719 GitHub Java projects and measured the time taken to complete the task. To create the code base of GitHub Java projects, we downloaded projects that received at least one star to avoid trivial repositories. We obtained 130,719 projects ranging from 29,465 stars to 1 star in January 2018. The most-starred project is RxJava (29,465 stars), followed by java-design-patterns (27,578 stars) and Elasticsearch (27,385 stars).

We simulated the scenario of maintaining a Siamese GitHub search index when the top three most-starred projects have a new version release. We started by adding

Table 7.13: GitHub projects used for incremental update

Project	#Releases	Average (Min, Max)		
		Size (MB)	Files	SLOC
RxJava	153	7 (0.4, 16)	582 (1, 1.5K)	82K (3, 244K)
java-design-patterns	13	15 (11, 18)	787 (479, 989)	15K (192, 26K)
Elasticsearch	214	62 (10, 145)	3.7K (1.2K, 5.6K)	399K (87K, 720K)

all the 130,719 Java projects into Siamese index one project at a time at method-level using incremental addition with the minimum method size of 10 lines (the preferred size of clone detection in large-scale code corpora [Sajnani et al., 2016]). The indexing took two weeks to finish, and the complete GitHub index contained 8.7 million code fragments with the size of 62 GB.

Then, we downloaded all the available releases of RxJava, java-design-patterns, and Elasticsearch to perform incremental version updates. We choose the three most-starred projects due to their popularity which reflects their chance of being searched for code. As displayed in Table 7.13, the number of releases and the size of each project varied. Elasticsearch had the highest number of 214 releases, followed by RxJava (153), and java-design-patterns (13) and also had the biggest size on average (62 megabytes), followed by java-design-patterns (15 megabytes) and RxJava (7 megabytes).

For each project, we repeated the process of updating the project’s releases from the oldest to the newest version by performing deletion of the current existing release stored in the index followed by addition of the next release to the index. For each update (i.e., deletion/addition) made to the Siamese’s index, we measured the time required to finish the task. The results are shown in Figure 7.10. The average time of updating java-design-patterns, which was the smallest of the three projects, took 6.6 seconds on average (median 6.5s, max 8.2s). For RxJava, the average project update time was 17 seconds (median 12.8s, max 51.1s). For Elasticsearch, which was the biggest projects and had the largest amount of revisions, the time Siamese took to update the index varied from 20 seconds to approximately 2 minutes with the average of 73 seconds (median 82.1s). The results show that Siamese’s incremental update could save the time to prepare the search index of

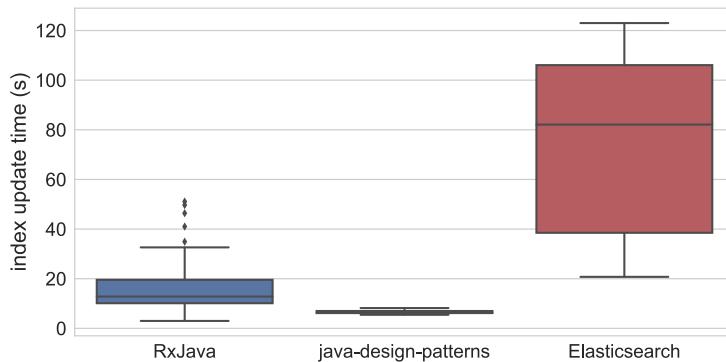


Figure 7.10: Incremental index update time

130,719 GitHub projects when a new version appears from 40,320 minutes (2 weeks) to 2 minutes.

To answer RQ5, Siamese incremental update efficiently handled the changes in software releases and dramatically decreased the index preparation time.

7.7 Threats to Validity

There are some potential threats to validity in this chapter. We separately discuss them in three aspects: internal, construct, and external validity.

7.7.1 Internal and Construct Validity

We carefully chose the data sets for our experiment and Siamese was evaluated on multiple data sets to cover several types of cloned code and to alleviate the evaluation bias. We compared the tools' performance based on three standard measurements of precision at 10, MAP, and MRR from information retrieval. Nevertheless, in some situations, other measurements may be required and might not produce the same results. We compared Siamese to seven state-of-the-art clone detectors on the default and the optimal configurations but we might not cover all the tools' parameters. Moreover, the n -gram sizes and the query reduction thresholds were derived from the Bellon corpus and may be subjective to the clones in the corpus but we mitigated the issue by avoiding using Bellon corpus in the evaluation data sets to avoid configuration bias. For the query reduction thresholds, we confirmed the findings with another corpus (Qualitas) and observed the same result. The manual validation of clone search results was carefully performed but

may still be subject to manual judgement and human errors. The MRR and precision at 10 used to measure Siamese’s precision differ from the precision typically used in code clone detection by validating only the top n clone results. They may not reflect the precision score which is based on the total number of returned results.

7.7.2 External Validity

Our multi-representation, query reduction techniques, indexing and searching performance of Siamese are evaluated with Java and may not be generalised to other languages. However, we design the Siamese’s architecture to work with other programming languages by plugging in a new tokeniser and code normaliser module. The indexing and querying performance of Siamese and SourcererCC were measured on a desktop computer and may not represent their performance on other computers with different specifications or a cluster of multiple Elasticsearch instances. The criteria for selecting the GitHub projects for incremental update is based on the stars and may not be generalised to other Java projects.

7.8 Chapter Summary

This chapter presents the architecture of a scalable and incremental clone search approach using multiple code representations and its implementation as a tool called Siamese. Siamese offers 95% and 99% mean average precision on the OCD and the SOCO data set respectively and also offers high recall for all clone types in the BigCloneBench data set. Furthermore, the tool provides scalability by returning clone search results in less than 8 seconds even on the largest data set of 365 million lines of code. The technique supports incremental index update that allows fast update to the existing index without a need to recreate the index from scratch.

The next chapter will discuss three applications of Siamese including a replication study of online code clones, software license analysis between code on Stack Overflow and GitHub projects, and recommending tests for reuse.

Chapter 8

Applications of Siamese

This chapter discusses the applications of Siamese to facilitate software development and research with three use cases including online code clone detection, clone search with automated license analysis, and integration of Siamese into a study of automating the reuse of tests.

8.1 Online Code Clone Detection on Stack Overflow

We replicated the study of online code clone detection in Chapter 4 between Stack Overflow Java accepted answers and the Qualitas corpus using Siamese, and compared the results to the existing clone results by Simian and SourcererCC. We used the same data sets of 72,365 Stack Overflow Java code snippets and 111 open source Java projects in Qualitas, and followed the same experimental framework to detect online code clones as shown in Figure 4.3 in Chapter 4, except that we did not need to partition the clone detection into multiple runs, thanks to the scalability of Siamese. The Qualitas corpus was added to Siamese index and the Stack Overflow code snippets were used as the queries. The configurations are shown in Table 8.1. We configured Siamese to consider methods with at least ten lines for the search, which resulted in 71,348 queries out of 149,664 methods in the 72,365 Stack Overflow snippets. We limited the result size at 100 code snippets per query.

Table 8.1: Siamese execution on Stack Overflow and Qualitas corpus

Snippets	Queries	Result size	Exec. time	Per query	Clone pairs (80% sim.)
72,365	71,348	100	1h 55m	0.10s	1,088

8.1.1 Similarity Threshold

Siamese is a clone search engine which returns a ranked list of clones based on relevance scores between the query and the retrieved code fragments. The original Siamese has no cut-off threshold to decide whether a retrieved code fragment is a cloned fragment of the query or not. This is desirable behaviour for a search engine because the user will look at only the top n results, but not for a clone detector that the user wants a comprehensive list of clones. To be able to compare the clone results of Siamese to Simian's and SourcererCC's, we adapted Siamese to incorporate a similarity measure called n -gram token ratio as the clone similarity threshold.

N -gram Token Ratio (NTR) is an n -gram based similarity measure specifically invented for Siamese. It is applied during search time. Siamese computes an NTR similarity score based on the number of tokens in the query that match with tokens in the indexed fragments. It is similar to Jaccard similarity on n -gram tokens, except that the similarity score is purely based on the query tokens instead of a union of tokens from the two code fragments. An NTR similarity score between a query Q and a code fragment F is computed as follows.

$$\text{Sim}_{\text{NTR}} = \frac{|\text{T}_Q \cap \text{T}_F|}{|\text{T}_Q|} \quad (8.1)$$

where T_Q represents a set of n -gram tokens in Q and T_F represents a set of n -gram tokens in F . Since Siamese uses four code representations for its clone search, the similarity score is applied to each of the four representations. Given a similarity threshold, Siamese retrieves only code snippets that offer an NTR score equal to or higher than the threshold on all four representations. The NTR score is applied when the search is performed. Only code fragments that contain enough tokens to reach the defined NTR similarity threshold are retrieved. This method effectively

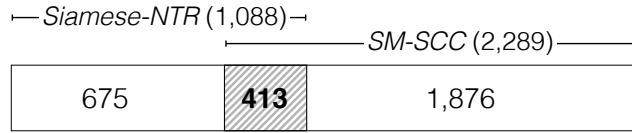


Figure 8.1: A comparison of Siamese-NTR clone pairs to the previous results by Simian and SourcererCC (SM-SCC)

Table 8.2: The 413 SM-SCC online clone pairs that are found by Siamese

QS	SQ	EX	UD	BP	IN	NC
125	1	111	64	112	0	0

prunes unrelated code fragments and results in fast query response time. In addition, the NTR is a simple token-based and language agnostic similarity measure. Thus, it supports an analysis of any programming language and also works with incomplete code fragments.

8.1.2 Results

As shown in Table 8.1, Siamese with NTR (Siamese-NTR) took approximately 2 hours to complete the clone detection, and the average clone search time per query is 0.10 seconds. We set the similarity threshold at 80% to be similar to the setting of SourcererCC’s clone similarity and obtained 1,088 clone pairs. Then, we compared the clone candidates to the existing clone results reported by Simian and SourcererCC (denoted SM-SCC) in Chapter 4. To find common clones between the new results from Siamese and the existing 2,289 SM-SCC clone pairs, we employed the clone matching method used in Chapter 4 by applying the Bellon’s ok-match clone agreement with the threshold t of 0.5.

8.1.2.1 Common Clone Pairs:

The comparison results are displayed in Figure 8.1. There were 413 common clone pairs between the SM-SCC results and Siamese-NTR. The common pairs spread across several clone patterns of QS, SQ, EX, UD, BP, IN, and NC as shown in Table 8.2. Siamese-NTR reported 125 Qualitas→Stack Overflow (QS) clone pairs out of the 153 discovered QS pairs by Simian and SourcererCC and one Stack Overflow→Qualitas (SQ) clone pair. It reported 111 external sources→Stack

Overflow (EX) pairs, 64 unknown direction (UD) pairs, and 112 boiler-plate (BP) pairs. It did not report any inheritance/interface (IN) or non-clone pair (NC).

8.1.2.2 Distinct Clone Pairs:

Siamese-NTR discovered 675 clone pairs that were not found before and also missed 1,876 clone pairs in the previous results (see Figure 8.1). To gain insights into the clone pairs that were found only by Siamese, we performed a manual investigation. The thesis author manually checked 100 randomly selected clone pairs from Siamese-NTR-only, the number of statistically significant sample with 95% confidence level and $\pm 10\%$ confidence interval. The manual clone validation reported 73 true clone pairs and 27 false clone pairs.

By applying the 80% NTR similarity to the four code representations, we forced Siamese to discover clones that were strictly similar. However, we did find some interesting clone pairs due to the NTR similarity computation. Since the n -gram token ratio is computed based on the number of tokens *in* the query, we found that Siamese-NTR could locate clones of the query inside another method. We call them *contained* clone pairs.

An example of the contained clone pairs is shown in Figure 8.2. The `addMouseListener()` method in the Stack Overflow answer ID 4151399 was reported as a clone fragment of the method `buildGUI()` from `AboutDialog.java` file from DrJava project although the query matched with only a segment of code inside the `buildGUI()` method. Looking closely into the cloned region between the two clone fragments, we made two observations. First, the object name differed. The first clone fragment contained an object called `component` while the second clone fragment contained a `drjava` object. The Siamese-NTR query could match them because of Type-2 clone representation that allowed variable renaming. Second, the two clone fragments contained a different ordering of the statements. They were reported by Siamese-NTR because the code representations based on n -grams allowed partial matching of statements.

Moreover, we also randomly looked at a few Simian-SCC-only clone pairs (50) to see why Siamese did not report them. We found many clones that were

```

/* query 4151399_1.java */
component.addMouseListener ( new MouseListener() {
    public void mouseClicked ( MouseEvent e ) {
    }
    public void mouseEntered ( MouseEvent e ) {
    }
    public void mouseExited ( MouseEvent e ) {
    }
    public void mousePressed ( MouseEvent e ) {
    }
    public void mouseReleased ( MouseEvent e ) {
    }
} );

/* drjava/ui/AboutDialog.java */
public void buildGUI ( Container cp ) {
    cp.setLayout ( new BorderLayout() );
    JLabel drjava = createImageLabel( DRJAVA, JLabel.LEFT );
    if ( drjava != null ) {
        drjava.setBorder ( new CompoundBorder (
            new EmptyBorder(5,5,5,5), drjava.getBorder()) );
        drjava.setCursor(new Cursor(Cursor.HAND_CURSOR));
        final String url = "http://drjava.org/";
        drjava.setToolTipText ( url );
        drjava.addMouseListener ( new MouseListener() {
            public void mousePressed ( MouseEvent e ) { }
            public void mouseReleased ( MouseEvent e ) { }
            public void mouseEntered ( MouseEvent e ) { }
            public void mouseExited ( MouseEvent e ) { }
            public void mouseClicked ( MouseEvent e ) {
            ...
            // 22 more lines
            ...
        }
    }
}

```

Figure 8.2: A contained type-3 clone pair reported by Siamese-NTR

similar but their similarity probably lower than our defined thresholds of 80%. In a few cases, they were Type-2 clones but missed by Siamese-NTR. It is because we equally applied 80% similarity to the four representations, and the r_1 , i.e., Type-1, representation rejected the clones. In this case, we should give a lower similarity threshold for the r_1 and r_2 representation and only maintain the 80% similarity threshold for r_0 and r_3 . Moreover, we observed several Simian-SCC clone pairs that were missed by Siamese because they spanned over multiple methods. They were detected by Simian because Simian only performed line matching to find clones. Since Siamese tried to parse the code into methods when possible, it could not detect this kind of clones.

Table 8.3: The data sets in the case study

Data set	Files	SLOC
Stack Overflow	72,365	1,840,581
GitHub	1,193,478	106,481,517

8.1.3 Discussion

Our replication of online code clone detection between Stack Overflow and Qualitas corpus using Siamese shows that the tool can be applied for fast searching of online code clones. The clone results after applying the n -gram token ratio (NTR) similarity measure have some overlaps with the existing clone pairs reported by Simian and SourcererCC and some distinct clone pairs only reported by Siamese. We manually checked the pairs reported by Siamese only and found that many of them are true positive pairs. At the same time, Siamese suffers from some false negatives. There were clone pairs that were reported by either Simian or SourcererCC that Siamese could not locate. This is possibly caused by the known problem of clone detection tools' configurations [Wang et al., 2013b].

8.2 Clone Search with Software License Analysis

This section illustrates an example of using Siamese for a large-scale exploratory study of clones that are shared between repositories and their license compatibility. An et al. [2017] performed a study of clones between Stack Overflow and 399 Android apps and their ramifications of license incompatibility. Their clone detector, NiCad, did not scale to the full data set and had to be executed in 100 smaller runs. Our study leverages the scalability of Siamese to do a similar study on a larger scale of Stack Overflow and 16,738 GitHub projects in a single run. The data sets used in this study consists of (1) Java code snippets on Stack Overflow and (2) Java source code in GitHub projects. The statistics of the two data sets are shown in Table 8.3. For GitHub, we downloaded Java projects with at least ten stars and obtained 16,738 projects. For Stack Overflow, we reused the 72,365 extracted code snippets from Java accepted answers employed in the previous case study.

To be able to check for license incompatibilities similar to the study by An et al.

[2017], Siamese was extended to support automatic software license identification using pattern matching¹, so that a manual investigation of software license is reduced to only the clone pairs that have incompatible licenses. We built a database of software license patterns by studying the list of 33 software license types on GitHub², reading the text in each license statement, and manually preparing the patterns. During the execution, Siamese identifies software license in a software project using a two-step approach. First, it reads a dedicated license file LICENSE or LICENSE.txt at the root level of each GitHub project and matches it with the license patterns in its database to detect the license at project-level. Second, Siamese reads a license statement on the top of each Java source code file and performs pattern matching of the license at file-level. When there is a conflict between the file-level and the project-level license, Siamese prefers the finer-grained file-level one. If the tool cannot identify the license, it reports unknown to flag that a manual validation is needed. Moreover, we configured Siamese to apply the n -gram token ratio (NTR) similarity of 100% to every query to make sure that we discovered only exact-match clones. Since Siamese supported incremental indexing, we sequentially indexed the projects one at a time. This also facilitated the project-based license identification that each project had to be analysed individually. The Siamese index, after analysing all the projects, contained 2,639,565 methods with an index size of 25.6 gigabytes. The indexing with license identification of GitHub projects took one day and twelve hours.

In the query phase, each code snippet from Stack Overflow was used as a query with a results size of 100. The search for clones with similarity computation between the two data sets took 1 hours and 57 minutes to complete.

8.2.1 Results

We initially set the minimum of 10 lines for clone size since it was recommended for a large-scale clone detection to get rid of trivial clones [Sajnani et al., 2016]. With the minimum of 10 lines, we retrieved a large number of clone candidate pairs.

¹We also tried integrating Ninka [German et al., 2010], a license identification tool, into Siamese but found that it dramatically slowed down the indexing and querying time.

²GitHub license type: <https://help.github.com/articles/licensing-a-repository>

Table 8.4: GitHub projects with the highest no. of clones

Project name	Stars	Clone pairs
google/j2objc	4,981	17
biblelamp/JavaExercises	34	13
xamarin/XobotOS	1,278	11
amirmehdizadeh/JalaliCalendar	51	9
javajavadog/guideshow	85	7
Odinvt/react-native-lanscan	16	7
aosp-mirror/platform_frameworks_support	1,253	5
osglworks/java-tool	16	5
dropbox/hackpad	3,085	5
ibrahimbalic/AndroidRAT	37	5

However, after manually investigating a few sampled clone pairs, we still found several trivial clones such as `equals` methods or generated GUI-related code. These trivial clones had the size of around 10 to 20 lines, so we increased the minimum clone size to 20 lines. With the larger minimum clone size, 378 clone pairs were reported. This is the lower bound of the number of clone candidate pairs since we might also get rid of true positive clone pairs that were smaller than 20 lines. Nonetheless, false negatives (i.e., not reporting a clone pair while it is actually a clone pair) are preferred over false positives (i.e., reporting a clone pair while it is actually a non-clone pair) in this case of license violation checking.

We compiled a list of 10 projects having the highest number of clones as shown in Table 8.4. The Google’s J2ObjC (4,981 stars), which is a command-line tool that translates Java to Objective-C code, has the highest number of 17 clone pairs. The second is `JavaExercises` project (34 stars), which contains a lot of Java programming examples, with 13 clone pairs followed by `XobotOS`, Android porting from Java/Dalvik to C#, (1,278 stars) with 11 clone pairs; `JalaliCalendar`, a Java Persian calendar library, (51 stars) with 9 clone pairs; `guideshow` (85 stars – 7 pairs); `react-native-lanscan` (16 stars – 7 pairs); AOSP Framework Support Library (1,253 stars – 5 pairs); `java-tool` (16 stars – 5 pairs); Dropbox’s `hackpad` (3,085 stars – 5 pairs); and `AndroidRAT` (37 stars – 5 pairs). We did not confirm the direction of cloning. However, after looking at the numbers, we observed an interesting patterns: high and low stars projects both have high numbers of clones,

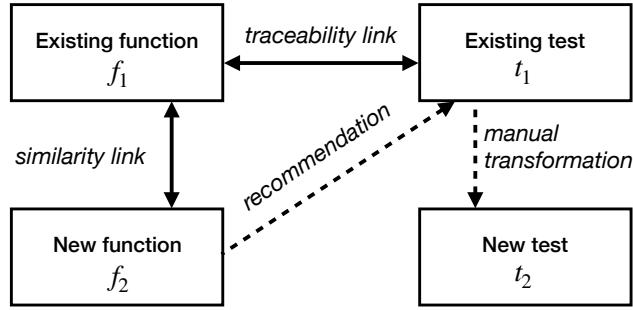
Table 8.5: License comparison of the clones

License	Stack Overflow	GitHub	Frequencies
Same license	None	None	127
	Apache-2.0	Apache-2.0	4
<i>Total</i>			<i>131</i>
Different license	None	Apache-2.0	139
	None	GPL-2.0	32
	None	MIT	27
	None	GPL-3.0	12
	None	Apache	10
	None	BSD-2-Clause	6
	None	BSD-3-Clause	5
	None	LGPL-3.0	5
	None	AGPL-3.0	4
	None	Artistic-2.0	2
	None	Unknown	2
	Unknown	CC0-1.0	1
<i>Total</i>			<i>247</i>
<i>Grand total</i>			<i>378</i>

which possibly indicate the direction of cloning. We left an investigation for future work.

Siamese reported the same license for 131 pairs, and different license for 247 pairs. We further analysed the licenses in the clone pairs and the results are displayed in Table 8.5. For the same license, 127 clone pairs do not have a license statement and 4 pairs have the Apache-2.0 license. On the other hand, 65% of the clone pairs with different licenses (247 out of 378) contain no license on Stack Overflow while having a license on GitHub. The three highest number of clone pairs have: 1) no license on Stack Overflow but Apache-2.0 license on GitHub (139 pairs); 2) no license on Stack Overflow and GPL-2.0 license on GitHub (32 pairs); and 3) no license on Stack Overflow but MIT license on GitHub (27 pairs).

Although we did not confirm the violations of software license, the findings from the study show that we can use Siamese to locate potential candidates of clones

**Figure 8.3:** The overview of RELATEST

with software license incompatibility, which save the time for a human investigator.

8.2.2 Discussion

The study demonstrated an application of Siamese to efficiently and effectively find clones which potentially violate software licenses. Siamese found a number of clone pairs between Stack Overflow and GitHub projects that the code were exactly matched but had different software licenses. These clone pairs with different licenses may or may not create licensing conflicts depending on the direction of cloning, which requires a further thorough investigation and is beyond the scope of this paper.

8.3 Automating the Reuse of Tests

This section shows an application of Siamese to facilitate the reuse of existing test cases. White et al.³ present an approach, called RELATEST, to extract test-to-code traceability links and use the discovered links to recommend tests to new and untested methods. Siamese has been used as a code similarity tool in the approach.

The main idea of the RELATEST approach is illustrated in Figure 8.3. RELATEST works on a software project with some existing unit tests. From the diagram, the RELATEST tool establishes a link between an existing function f_1 and an existing unit test case t_1 using 1) naming convention (NC) between the test name and the function name (e.g., `add` and `testAdd`) and 2) the Last Call Before

³R. White, J. Krinke, E. Barr, C. Ragkhitwetsagul, F. Sarro, and A. Mariam, *Exploiting test-to-code traceability links for reuse*, Submitted to the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE '18)

Assert (LCBA) technique [Rompaey and Demeyer, 2009]. Once a traceability link has been established, we can recommend the test t_1 to a function that is newly written or does not yet have a test based on a similarity between f_1 and the function. If a new function f_2 is similar enough to f_1 (based on some definition of similarity), we recommend the test t_1 from f_1 to f_2 . The developer then copies and adapts the recommended test t_1 to fit within the new environment required to test f_2 . This technique can help to save the developer's time on writing a new test from scratch.

8.3.1 Searching for Similar Functions using Siamese

The actual implementation of the RELATEST idea is depicted in Figure 8.4. T denotes all tests from the code corpus, F denotes all functions from the code corpus, L denotes the set of traceability links between T and F , f_q is the query function to look for other similar functions, $S(f_q)$ is the list of functions that are similar to the query function, and $R(f_q)$ is the list of test recommendations for f_q . The main components of RELATEST, including traceability link establishment module and recommendation module, were implemented by Robert White, a PhD student in CREST, UCL under the supervision of Dr. Jens Krinke. Siamese was chosen as a code similarity tools in the query processor module. The author has collaborated with Robert White to integrate Siamese into RELATEST.

The test recommendation works as follows. RELATEST starts by analysing a given code corpus and generating traceability links between functions and tests in the corpus. The user of RELATEST gives a new function f_q that he or she wants to get test recommendations. Then, f_q is sent to Siamese in the query processor module to search for similar functions $S(f_q)$. RELATEST reads the Siamese search results and consult with the database of traceability links and finally returns a list of recommended test cases $R(f_q)$ back to the user.

8.3.2 Results

Robert White and the author evaluated the performance of RELATEST based on three Java systems: JFreeChart, CCollections, and Marc4j. The quality of test recommendations was measured in two scenarios: within-project and cross-

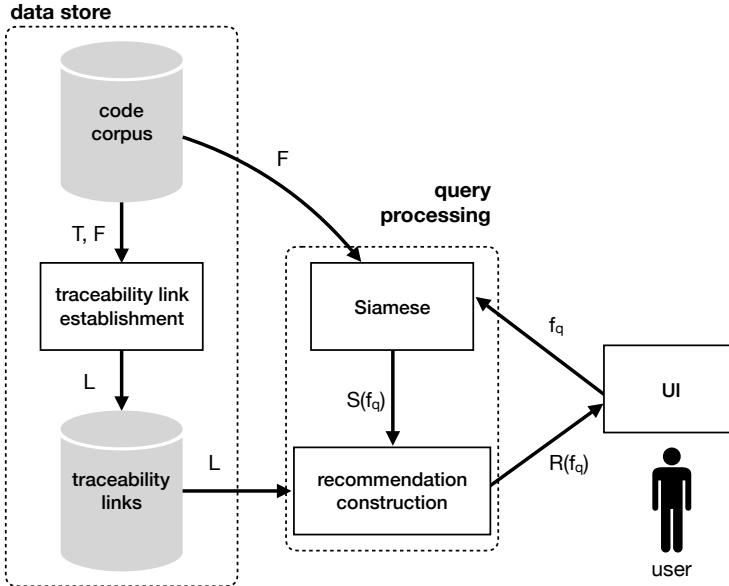


Figure 8.4: An implementation of RELATEST with Siamese as a query processor.

project. The within-project recommendations use a query function from one project to recommended tests from within the same project, while the cross-project recommendations use a query function from one project and recommend tests from another project. From the three systems, we randomly sampled 10 functions with tests as query functions. Siamese was executed to build a search index on a base project containing tests to recommend. By giving the 10 sampled functions as queries to RELATEST, Siamese queried its search index and returned 10 ranked lists of similar functions to RELATEST. Then, RELATEST consulted the traceability link database to find tests of the retrieved similar functions. The tests were recommended to the human investigators to confirm their reusability.

We manually validated the RELATEST recommendations using the error measures shown in Table 8.6. The evaluation is performed on all recommendations and per-query recommendations. For the all recommendation evaluation, we counted the number of true and false positive recommendations by manually looking at the recommended tests and decided whether they can be reused or not. The precision score was computed based on the number of true positives over the total number of recommendations. For the per-query evaluation, we counted the number of recommendation lists that contained at least one true positive recommendation as

Table 8.6: Ranked List Validation Methods.

Statistic	Description
<i>All Recommendations</i>	
TP	The number of true positive recommendations
FP	The number of false positive recommendations
Prec	The number of true positives over the total number of recommendations, i.e., $\frac{ TP }{ Recommendations }$
<i>Per-Query</i>	
TP_L	The number of lists that contain at least one true positive recommendation
FP_L	The number of lists that contain no true positive recommendations
$Prec_L$	The number of true positive lists over the total number of samples, i.e., $\frac{ TP_L }{ Samples }$
MRR	The mean reciprocal rank of the lists
P@5	The average precision at rank 5 of the lists

Table 8.7: Manual validation of recommendation lists (within-project)

Tool	Queries	All			Per-Query				
		TP	FP	Prec	TP_L	FP_L	$Prec_L$	MRR	P@5
JFreeChart	10	25	10	0.71	5	5	0.50	0.50	0.50
CCollection	10	12	16	0.43	6	4	0.60	0.60	0.46
Marc4j	10	0	24	0.00	0	10	0.00	0.00	0.00

true positive and counted the number of recommendation lists that contained no true positive recommendation as false positive. Then, we computed precision, mean reciprocal rank (MRR), and precision-at-5 (p@5) scores.

Table 8.8: Manual validation of recommendation lists (cross-project)

Tool	Queries	All			Per-Query				
		TP	FP	Prec	TP_L	FP_L	$Prec_L$	MRR	P@5
JFreeChart	10	25	10	0.71	5	5	0.50	0.50	0.50
CCollection	10	13	19	0.41	6	4	0.60	0.60	0.46
Marc4j	10	1	33	0.03	1	9	0.10	0.10	0.10

8.3.2.1 Within-Project Recommendations

The manual validation results are shown in Table 8.7. Siamese gave a precision score of 0.71 and 0.43 for JFreeChart and CCollections respectively over all recommendations. The precision for Marc4j was zero since due to no true positive recommendation. Considering the per-query evaluation, the test recommendations were useful in half of the cases for JFreeChart (5 TP_L and 5 FP_L). The MRR score was 0.50, which meant half of the time the list contained correct recommendation on the top. The 0.50 precision at 5 meant that, on average, the top 5 results contained 50% true positives. A similar observation was found for CCollections, we observed a slightly higher number of true positives (6 TP_L and 4 FP_L). The project offered a slightly higher precision and MRR scores of 0.60, but a lower precision at 5 of 0.46. Since Marc4j did not produce any useful recommendation, its per-query scores were all zero.

8.3.2.2 Cross-Project Recommendations

The results were very similar to within-project recommendations as shown in Table 8.8. We observed a precision score of 0.71 and 0.41 for JFreeChart and CCollections respectively over all recommendations. Interestingly, we found one true positive recommendation out of 34 recommendations for Marc4j. The precision of recommendation on Marc4j was 0.03. Considering the per-query evaluation, similar findings were observed with MRR scores of 0.50 and 0.60 for JFreeChart and CCollections, and 0.10 for Marc4j. The precision at 5 scores were 0.50, 0.46, and 0.10 respectively.

8.3.3 Discussion

We have shown that Siamese can be applied to automate a test recommendation task. With the nature of Siamese as a query-based code search engine, it fits well with the RELATEST technique where one wants to find functions similar to the query function and only top n ranked results are needed. Moreover, the scalability of Siamese will be valuable when large-scale code corpora, such as GitHub, are required for the recommendations. The evaluation shows that some of the

recommended tests based on Siamese search results were useful and both within- and cross-project recommendations produced very similar results. There were good recommendations for JFreeChart and decent recommendations for CCollections. However, Siamese did not perform well on Marc4j. We found only one case of a useful recommendation for Marc4j on the cross-project recommendation.

Since this is an ongoing project, we aim to improve the test recommendation quality further by optimising Siamese's parameters (e.g., query reduction thresholds) to tailored to the token distributions in the analysed corpus as well as increasing the precision of the traceability establishment and the test recommendation module.

8.4 Chapter Summary

This chapter illustrates the applications of Siamese to software engineering research. The chapter has shown that Siamese clone search technique is general and can be adapted to a wide range of problems, such as online code clone detection, software license analysis, or automated test recommendation.

The next chapter is the last chapter of the thesis. It will summarise the contributions of this thesis and discuss the future work.

Chapter 9

Conclusion and Future Work

Large-scale source code data facilitate code cloning and, at the same time, complicate the detection of such cloning. This thesis has established the existence of online code cloning, which occurs between Stack Overflow Q&A website and software projects. The online surveys reveal that Stack Overflow users are aware and concerned of the ramifications of online code cloning to and from Stack Overflow, including outdated code and software license violations. The thesis has analysed online code clones between Stack Overflow and Qualitas corpus and found outdated and potentially license-violating cloned code snippets, which may be harmful for reuse.

To support the detection of online code clones, a scalable clone search approach is a necessity. The thesis has presented a scalable and incremental code clone search technique, called Siamese, and shown that it is suitable for clone search in large-scale source code data. The use of multiple code representations and query reduction allows Siamese to flexibly detect clones from Type-1 to Type-3. Each clone search query costs only a few seconds. The thesis has shown that Siamese can be deployed to tackle several research problems in software engineering.

In this way the analysis of code similarity and code clones on large-scale source code data can be efficiently carried out. We foresee that scalable code similarity and clone search approaches and tools will be valuable in tackling several code similarity-related research problems in the near future.

9.1 Summary of Achievements

The main goal of this thesis is *to study code cloning in large-scale source code data and develop a scalable clone search approach to address challenges from such cloning*. Towards that goal, this thesis has produced the following contributions.

Online Code Clones

Online code cloning and its side-effects of outdated code and software license incompatibility are established and investigated in Chapter 3 and Chapter 4. The two online surveys show that Stack Overflow answerers and visitors are aware of outdated code snippets. In contrast, they are not aware or not concerned about the violations of the original software license of code in the answers. The empirical clone study between Stack Overflow and 111 Java open source projects confirms the survey results. There are 2,063 manually confirmed clone pairs found between the two sources. Several clones are copied from the open source projects or external sources to Stack Overflow, and many of them (100 clone pairs) are outdated. 214 code snippets could potentially violate the license of their original software, and they occur in 7,112 projects on GitHub.

Framework for Comparing Code Similarity Tools and Clone Search Techniques

Chapter 5 presents the OCD framework that is created for evaluating code similarity and clone search tools based on a data set with pervasive code modifications and a data set with boiler-plate code. The thesis uses the framework to study the strengths and weaknesses of 34 state-of-the-art code similarity tools and techniques. The results show that the dedicated code similarity tools, such as clone detectors and plagiarism detectors, outperform more general techniques of string matching or normalised compression distance. However, there are a few string similarity techniques, including Jaccard and Sorensen-Dice on n -gram tokens, and Python's FuzzyWuzzy, and Difflib libraries, that offer higher performance than the dedicated tools on both clone detection and clone search scenarios. The empirical study also confirms

that the tools' optimised configurations are biased to a data set and are not recommended to be reused. The framework can be used as a benchmark for evaluating future code similarity tools.

Enhancing Code Clone Detection using Compilation/Decompilation

The use of compilation and decompilation as a code normalisation process for code clone detection is first investigated in Chapter 5 and later evaluated in more detail in Chapter 6. The results in Chapter 5 shows that compilation followed by decompilation increases the F1 scores of 34 code similarity tools with statistical significance. The follow-up study in Chapter 6 discover similar findings that, when applied to a software project, compilation and decompilation allow a clone detector to find more clones, especially challenging ones that are missed on the original source code. The technique increases the clone detector's recall without sacrificing precision.

A Scalable and Incremental Code Clone Search Approach

The thesis develops Siamese, a scalable and incremental code clone search approach to tackle the challenge of locating online code clones in large-scale source code data and studying their issues. Using the observations from a comparison of code similarity analysers in Chapter 5, Siamese adopts n -gram of normalised code tokens as the intermediate code representation. It uses multiple code representations and query reduction techniques to accurately search for clones of Type-1 to Type-3. The tool is scalable to a large code corpus of 365M SLOC and allows incremental updates to its search index. Chapter 8 discusses three applications of Siamese including online code clone detection, software license analysing of online code clones, and automated test reuse.

9.2 Summary of Future Work

There are now several scalable code clone detection tools available [Sajnani et al., 2016, Saini et al., 2018, Kim et al., 2018] including Siamese. Their large-scale empirical studies reveal interesting findings that could not be achieved using

classical tools (e.g., Yang et al. [2017], Saini et al. [2016b], Lopes et al. [2017]). Yet, there remains several future work to be done in the area of code similarity and clone search in large-scale source code data. We discuss some potential future work below.

Automated Code Cloning Direction Detection

As shown in this thesis, in order to validate the direction of online code cloning, we still rely on manual validation to read the Stack Overflow posts, understand the context, and use multiple hints based on human judgement (e.g., comments in the code, natural text in the question and answers, date/time of the posts) to conclude that the code snippets are actually copied from Qualitas or external sources to Stack Overflow. This is a burdensome and time-consuming process. The future work is to automate the code cloning direction detection. From our experience in this thesis, we found that code comments and the accompanied natural text on Stack Overflow were a great source of information to decide the direction of code copying. Thus, by using code clone detection to locate clone candidates and then applying information retrieval techniques, e.g., cosine similarity with tf-idf, we can rank the clone candidates based on the similarity of their project names (or classes) to the text in comments or natural text surrounding the clones in Stack Overflow posts. For example, a Stack Overflow answer containing the text “*Actually, you can learn how to compare in Hadoop from WritableComparator. Here is an example that borrows some ideas from it.*” must be ranked very high among the list of clone candidates of a code snippet from Hadoop since it contains two terms of the project name (Hadoop) and a class name (WritableComparator) in it. This technique will dramatically reduce the manual validation effort to establish the direction of cloning. The technique can also be used on Stack Overflow to flag that an answer has a high chance of copying from open source projects.

Multi-Modal Software Similarity Measurement

Most of the code similarity tools, scalable or not, locate similar pieces of code based on their source code. This includes our Siamese tool. Some studies try to include

other information such as natural text surrounding source code to locate more clones [Kim et al., 2018] or to establish a link between natural text and source code for code search [Ye et al., 2016]. With the multiple-code-representation concept in Siamese, we can include other types of information besides source code in the code search engine and open more possibilities of finding similar software artefacts. The technique can be used to search for similar applications using a combination of software requirements, documentation, and source code. Another possibility is to find similar code reviews based on a patch, reviewer name, and review text.

Code Clone Detection at Code Review Time

Code clone detection has been integrated into the software development process in several ways. Some software companies run a standalone code clone detector on their software project at a time they want to refactor them [Sajnai, 2016]. Some companies integrate the clone detector in their software lifecycle and study the clones in every revision (e.g., CQSE’s Teamscale solution). Some programmers are made aware of clones by their IDEs (e.g., JetBrains’s IntelliJ IDEA, SourcererCC-I Eclipse Plug-in [Saini et al., 2016a]). There are also tools from research that make the developer decide to or not to clone while they copy the code [Wang et al., 2012].

We see that code review is an appropriate stage for reporting clones. It has a few benefits over real-time clone detection in IDE or during commit time. First, by integrating code clone detection into code review, we can detect clones before they are merged into the code base. It shares the same outcome of preventing (or warning about) the creation of clones as the IDE-based and the commit-based clone detection. In addition, code review is a better time for clone detection than the post-mortem approaches, i.e., the approaches that identify clones after they already appear in software. Moreover, the code review is specifically designed for manual investigation of a patch. It is possibly better accepted by the programmers than clone detection in the IDE, which interrupts programmers while coding. We can study the awareness of programmers to code cloning and how code-review-time clone detection affects the software quality.

Code Clone Search As a Service

Siamese is built on top of Elasticsearch which supports distributed computing. By upgrading Siamese to a web service on multiple machines, we are enabled to offer a cloud-based clone search service for researchers and practitioners. The clone search system can be integrated into the software development process as an IDE plug-in or an automated code review assistant. Our clone search service will analyse code snippets that are being committed into the project repository or being reviewed by other programmers in the team and response back within seconds when it finds similar code fragments on the Internet. Thus, software companies or open source projects can perform a real-time check of code copying-and-pasting to avoid the issues from code cloning (e.g., bug propagation or license violations), or at least to be aware that such activities occurred in their software development process. In return, we will collect the usage history and probably the code fragments they submit to the system for a large-scale analysis of clone search.

Appendix A

Chapter 3: Answerer and Visitor Surveys

A.1 Open Comments from Stack Overflow Answerers

1. Sometimes you have to post code from official documentation, like in case of C#, code form MSDN is posted in the answer with added explanation.
2. Snippets on SO are usually for demonstrating a technique and therefore age well. If otherwise, I usually made them a gist, codebin or jsfiddle.
3. The real issue is less about the amount the code snippets on SO than it is about the staggeringly high number of software “professionals” that mindlessly use them without understanding what they’re copying, and the only slightly less high number of would-be professionals that post snippets with built-in security issues.
4. A related topic is beginners who post (at times dangerously) misleading tutorials online on topics they actually know very little about. Think PHP/MySQL tutorials written 10+ years after `mysql_*` functions were obsolete, or the recent regex tutorial that got posted the other day on HackerNew (<https://news.ycombinator.com/item?id=14846506>). They’re also

full of toxic code snippets.

5. Just to say: the reason that I very rarely check the license status is that the code I am posting is almost always my own or adapted from the question, or imported from an open source project that I have worked on and already know the license terms, or from my own company code that I can 100% say it is OK to post in public because I know our policies.
6. No. Code snippets are short and small enough that no IP can be in them.
7. The point of snippets are that they are trivial. I mostly write from scratch or copy-&-fix a snippet from the question. Most are illustrative or incomplete – they aren't of any value at all in isolation. They rarely take more than a few minutes to write, and it's usually harder to explain what they're doing in plain English. Where something is large enough to worth the effort of licensing then it's far too big for a snippet. In those cases I create a GitHub project (with a license) and link to it. I'd be wary of increased IP controls – I doubt there is value they could add to snippets, but they could create significant barriers to contributors, which would hurt the site.
8. I always try to see what kind of person is asking the question. If it is a student, I don't want to just hand out the answer; they will learn nothing from that. If, on the other hand, it's somebody looking for best practices or a clever trick, I'm not too worried about giving out the solution. In this case, chances are much higher that the person asking the question will go "Ahh, yes, of course!" and understand the question, whereas some students are more likely to mindlessly copy-paste the answer.
9. However, it is also a competitive site, so if your answer requires too much work to incorporate, it won't get accepted or upvoted. As a result, some people—I'm guilty of this myself, I'm sure—will hand out an answer willy-nilly that might solve the problem at hand, but in the long run be a disservice

to the person asking.

10. But I digress: My point is that it's sometimes better to describe a solution rather than just hand off a code snippet. However, since your project seems to pertain to copyright issues, I suppose that's not relevant to you.
11. I'm not sure it's possible to include a license in your questions/answers. I'm not sure what the legal ramifications would be if you tried it, since you already agreed to S.O.'s terms. This is a very interesting question and I look forward to hearing the results of your research.
12. I think it's important to realise the code snippets are designed to be very small, useful to illustrate concepts (10 lines or less). When you consider licensing laws of such a small amount of code, while technically may be violating a licence, in practice it would be nearly impossible to enforce such a claim.
13. SO code snippets are great but there is lot to improve . It's hard to edit and see output in so but website like jsfiddle , jsbin provide nice interface where code editing and output is easy to do.
14. Outer thing is in so lot of code snippets doesn't work because some users don't add libraries like angular, jquery. I think it's better if we can identify and ask user to auto inject relevant libraries.
15. Code snippets are usually just a few lines of code so it will be hard to enforce any copyright claims except when it is a method used for something company-specific (such as generating encryption keys). Regardless, since most of the code I write is specifically to answer a given question and having full knowledge of the license system used by Stack Overflow, it is entirely unimportant to concern myself with licensing the code provided. Also, code from MSDN documentation which I sometimes adapt and modify for answers are already in the public domain so it makes no sense re-licensing it.

16. On the matter of deprecation, I almost entirely use .NET which has got different versions of the framework. Therefore, code deprecation is not often a problem since what is deprecated on one version of the framework may be the only way of solving a given problem on an older version of the framework. I may also have to add that questions I tend to answer are about how to solve general coding problems so they are not usually subject to deprecation.
17. I think you're forgetting the fact that as a community, we want to share knowledge. Patents, copyright issues and so on – it's all just annoying. We're there to have fun and to share knowledge with people.
18. In the early days, the internet used to be full of free-for-all stuff without any licenses. Because of that, it was a fantastic tool to share knowledge and information on a vast scale. The remnant of this, open source, couldn't have existed without this!
19. Personally, I believe this “intellectual property” drive of the last decade is completely overrated. If you make something **substantial**, it's fine to be able to claim some ownership – but on snippets? It's like patenting the stuff you make in your free time in your shed... it doesn't make sense and just adds to the pile of legal bullshit imho.
20. No. I only put code on there that I have the right to (code I created or have permission to share). Adding the code is not an issue for me.
21. When I copy code it's usually short enough to be considered “fair use” but I am not a lawyer or copyright expert so some guidance from SO would be helpful. I'd also like the ability to flag/review questions that violate these guidelines.
22. The snippets are all small enough that I reckon they fall under fair use.

23. I always try and attribute the code I take from other places. I feel pointing back to the origin should be sufficient in terms of giving credit where it's due. Open source is about the sharing of ideas so that others can build on them. Education is a primary use case of open source in my opinion.
24. My only concern, albeit minor, is that I know people blindly copy my code without even understanding what the code does.
25. This survey may be inapplicable to me because I never copy code from existing projects.
26. Stack Overflow did an effort to apply a MIT license to all code snippets, while keeping CC-SA for the text content. Too bad they didn't succeed with it, as it would have solved many issues.
27. The main problem for me/us is outdated code, esp. as old answers have high Google rank so that is what people see first, then try and fail. That's why we're moving more and more of those examples to knowledge base and docs and rather link to those.
28. I've got no issues with code snippets on Stack Overflow, I think they are great. Any one using them should pay attention to details such as the date of the answer etc.
29. SO's license is not clearly explained when one registers or starts to answer questions.
30. No, most copied code snippets are so trivial that licensing them would be nearly impossible. It's also mostly modified version, where only some patterns are used.
31. Lot of the answers are from hobbyist so the quality is poor. Usually they are

hacks or workarounds (even MY best answer on SO is a workaround).

32. I think most example and explanatory snippets don't need a code-specific license. The CC license provides just fine. The examples either aren't copyrightable in the first place, or merely used as starting point (not used exactly as-is, not very different from reading an "All rights reserved" education book when learning programming, and "using" it in your career every day going forward). In addition, there is also the attitude of authors. Where I might care about attribution for distribution of my answer, the code within my answer is always Public Domain for me, meaning, I would never defend it. (I used to state that on my profile as well, but not in every post.)
33. Correctness and even syntax are often in doubt if I haven't had time to test the snippet end-to-end under the OP's conditions/environment.
34. It will be awesome if it becomes simple git repositories like github's gist.
35. Note that although I was not specifically of SO's licensing terms, I did have an in mind what those terms were likely to be. I have always made sure that there should be no reason that I should not share the code that I included in my replies.
36. It's an ESSENTIAL part of the site, it would NEVER work without such pieces of code. Also, given the snippets are very small in 99.99% of cases, legal aspects of this are inherently and pretty much always overlooked by the users.

A.2 Stack Overflow Answerer Survey: Google Forms

A survey of developers' experiences on answering Stack Overflow questions with code examples

Dear developer,

Since you are one of the top answerers on Stack Overflow, we hope you can help us with a not-for-profit study.

We are researchers in the Software Systems Engineering Group at University College London, UK. We are studying problems caused by outdated and license-violating code snippets on Stack Overflow. We have designed a survey to understand these problems and would be grateful if you would complete it.

The survey is completely anonymous and has 11 questions and should only take about 3-5 minutes to complete. We hope that you will complete the entire form but if you do not wish to continue, you can just quit the session and your input will be discarded.

The survey results will be used only for academic research purposes and we plan to release the results to Stack Overflow and in the form of academic papers and presentations.

This research project has been approved by the designated ethics officer in the Computer Science Department at UCL.

If you have any question, please feel free to contact me.

Chaiyong Ragkhitwetsagul
chaiyong.ragkhitwetsagul.14@ucl.ac.uk
<http://www.cs.ucl.ac.uk/staff/C.Ragkhitwetsagul>

* Required

1. How long have you been working on developing software? *

Mark only one oval.

- Less than a year
- 1 - 2 years
- 3 - 5 years
- 5 - 10 years
- More than 10 years

2. How frequently do or did you answer questions on Stack Overflow? *

Mark only one oval.

- Very Frequently (every day)
- Frequently (roughly 3-6 times a week)
- Occasionally (roughly once or twice a week)
- Rarely (roughly once or twice a month)
- Very Rarely (roughly once or twice a year)
- Never *Skip to question 11.*

Experience of Answering Stack Overflow Questions

This section aims to gain understanding on how developers choose code snippets to put as a solution on Stack Overflow.

3. How frequently do or did you include code snippets in your answers on Stack Overflow? *

Mark only one oval.

- Very Frequently (81--100% of the time)
- Frequently (61--80% of the time)
- Occasionally (41--60% of the time)
- Rarely (21--40% of the time)
- Very Rarely (1--20% of the time)
- Never (0% of the time) *Skip to question 11.*

Sources of the Stack Overflow Snippets

This section will gather information about the origins of the example snippets in Stack Overflow answers.

4. Where did the code snippets in your answers come from? *

Mark only one oval per row.

	Very frequently	Frequently	Occasionally	Rarely	Very rarely	Never
I copied them from my own personal projects.	<input type="radio"/>					
I copied them from my company's projects.	<input type="radio"/>					
I copied them from open source projects.	<input type="radio"/>					
I wrote the new code from scratch.	<input type="radio"/>					
I copied the code from the question and modified it for the answer.	<input type="radio"/>					
Others	<input type="radio"/>					

Concerns of Copying Code Snippets to Stack Overflow

This section aims to study concerns of the answerers regarding software licensing of their code snippets in Stack Overflow answers.

5. Were you aware, at the time of copying the code, that Stack Overflow apply Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) to content in the posts, including code snippets? *

Mark only one oval.

- Yes
- No

6. Do you usually include a software license in your code snippets on Stack Overflow? **Mark only one oval.*

- Yes, as a comment in the code.
- Yes, in a text surrounding the source code.
- Yes, both in code comments and text.
- No.

7. How frequently did you check the software license of the code snippets you copy to Stack Overflow if they conflict with Stack Overflow's CC BY-SA 3.0 license? **Mark only one oval.*

- Very Frequently (81--100% of the time)
- Frequently (61--80% of the time)
- Occasionally (41--60% of the time)
- Rarely (21--40% of the time)
- Very rarely (1--20% of the time)
- Never (0% of the time)

Problems from Code Snippets on Stack Overflow

This section will gather information of code snippets that were outdated after being copied to Stack Overflow.

8. Outdated code occurs when code snippets in your answers are no longer up-to-date with the latest version of the software you copied the code from. Have you ever been notified of outdated code in your Stack Overflow answers? **Mark only one oval.*

- Yes
- No

Frequency of Notifications**9. How frequently were you notified of outdated or deprecated code in your Stack Overflow answers? ****Mark only one oval.*

- Very frequently (81--100% of my answers)
- Frequently (61--80% of my answers)
- Occasionally (41--60% of my answers)
- Rarely (21--40% of my answers)
- Very rarely (1--20% of my answers)
- Never (0% of my answers) *Skip to question 11.*

Fixing Outdated Code

10. How frequently did you fix your outdated code on Stack Overflow? **Mark only one oval.*

- Very frequently (81--100% of the cases)
- Frequently (61--80% of the cases)
- Occasionally (41--60% of the cases)
- Rarely (21--40% of the cases)
- Very rarely (1--20% of the cases)
- Never (0% of the cases)

Additional feedbacks**11. Do you have any other concerns regarding answering Stack Overflow with code snippets?**

Powered by



A.3 Stack Overflow Visitor Survey: Google Forms

A survey of developers' experiences on reusing code snippets from Stack Overflow

Dear developer,

We are researchers in the Software Systems Engineering Group at University College London, UK. We are studying problems caused by outdated and license-violating code snippets on Stack Overflow. We have designed a survey to understand these problems and would be grateful if you would complete it.

The survey is completely anonymous and has 15 questions and should only take about 5-7 minutes to complete. We hope that you will complete the entire form but if you do not wish to continue, you can just quit the session and your input will be discarded.

The survey results will be used only for academic research purposes and we plan to release the results to Stack Overflow and in the form of academic papers and presentations.

This research project has been approved by the designated ethics officer in the Computer Science Department at UCL.

If you have any question, please feel free to contact me.

Chaiyong Ragkhitwetsagul
chaiyong.ragkhitwetsagul.14@ucl.ac.uk
<http://www.cs.ucl.ac.uk/staff/C.Ragkhitwetsagul>

* Required

1. How long have you been working on developing software?

Mark only one oval.

- Less than a year
- 1-2 years
- 3-5 years
- 5-10 years
- More than 10 years

Importance of Stack Overflow to Developers

This section aims to measure importance of Stack Overflow to developers when they solve programming tasks.

2. When you had a problem with your programming tasks, please rank in which order did you search for help (1 for the first, and 5 for the last, without a tie)? *

Mark only one oval per row.

	1st	2nd	3rd	4th	5th
Books	<input type="radio"/>				
Official documentations	<input type="radio"/>				
Stack Overflow	<input type="radio"/>				
Online repositories (e.g. GitHub)	<input type="radio"/>				
Others	<input type="radio"/>				

3. How frequently do or did you copy source code snippets from Stack Overflow? *

Mark only one oval.

- Very frequently (everyday)
 Frequently (roughly 3-6 times a week)
 Occasionally (roughly once or twice per week)
 Rarely (roughly once or twice a month)
 Very rarely (roughly once or twice a year)
 Never *Stop filling out this form.*

Reasons for Reusing Stack Overflow's Snippets

This section aims to understand why developers reuse code from Stack Overflow and their problems.

4. Why did you copy and reuse code snippets from Stack Overflow?

Mark only one oval per row.

	Strongly agree	Agree	Undecided	Disagree	Strongly Disagree
They are easy to find by searching the web.	<input type="radio"/>				
They solve problems similar to my problems with minimal changes.	<input type="radio"/>				
The context of questions and answers helped me understand the code snippets better.	<input type="radio"/>				
The voting mechanism and accepted answers helped filtering good code from bad code.	<input type="radio"/>				

5. Have you ever found any problems from reusing Stack Overflow code snippets? *

Mark only one oval.

- Yes
 No *Skip to question 10.*

Problems from Stack Overflow Code Snippets

This section aims to understand the problems from Stack Overflow snippets and whether the developers notify the answerers of the problems.

6. How frequently did you find problems from reusing Stack Overflow code snippets? *

Mark only one oval.

- Very frequently (81–100% of the reused snippets)
 Frequently (61–80% of the reused snippets)
 Occasionally (41–60% of the reused snippets)
 Rarely (21–40% of the reused snippets)
 Very rarely (1–20% of the reused snippets)
 Never

7. What were the problems? **Check all that apply.*

- Incorrect solution (the code claims to solve the problem in the question while it does not).
 Outdated solution (the code may work with some older versions of the library or API, but not the one you are using).
 Mismatched solution (the code solves the problem in the question but it is not exactly the right solution for your problem).
 Other: _____

8. How frequently did you report the problems back to the Stack Overflow discussion threads? **Mark only one oval.*

- Very frequently (81--100% of the time)
 Frequently (61--80% of the time)
 Occasionally (41--60% of the time)
 Rarely (21--40% of the time)
 Very rarely (1--20% of the time)
 Never (0% of the time) *Skip to question 10.*

Reporting problems of Code on Stack Overflow

9. How did you report the problems?*Check all that apply.*

- I down-voted the answer containing the problematic code snippet
 I wrote a comment saying that the code has problems.
 I contacted the answerers regarding the problems directly.
 I never report problems.
 Other: _____

Licensing of Code on Stack Overflow

This section will study awareness of developers regarding licensing of code snippets on Stack Overflow.

10. Were you aware, at the time of copying the code, that Stack Overflow apply Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) to content in the posts, including code snippets? **Mark only one oval.*

- Yes
 No

15. Could you please briefly explain what are the legal problems that you faced by copying code snippets from Stack Overflow?

Other Comments

16. Did you have any other problems from using code snippets from Stack Overflow?

Powered by
 Google Forms

Appendix B

Chapter 4: Outdated Code Snippets

Table B.1: 100 outdated cloned code snippets on Stack Overflow and their associated original projects

No.	Stack Overflow Post	Start	End	Qualitas Project	File	Start	End
1	9291241	1	11	apache-ant-1.8.4	Mkdir.java	19	29
2	12106623	1	16	apache-log4j-1.2.16	WriterAppender.java	45	61
3	18232672	42	63	apache-log4j-1.2.16	SMTPAppender.java	207	228
4	21734562	1	70	apache-tomcat-7.0.2	BasicAuthenticator.java	23	87
5	24404964	1	36	apache-tomcat-7.0.2	CoyoteAdapter.java	543	578
6	12617195	7	20	apache-tomcat-7.0.2	ServletFileUpload.java	12	25
7	10289462	1	45	apache-tomcat-7.0.2	JspRuntimeLibrary.java	252	296
8	24404964	14	31	apache-tomcat-7.0.2	CoyoteAdapter.java	557	573
9	8409971	65	86	apache-tomcat-7.0.2	GzipOutputFilter.java	47	68
10	16860613	55	74	eclipse_SDK	StyledString.java	55	74
11	2513183	106	159	eclipse_SDK	GenerateToStringAction.java	113	166
12	7504040	5	17	eclipse_SDK	ExternalResource.java	8	20
13	16860613	96	117	eclipse_SDK	StyledString.java	96	117
14	16860613	118	165	eclipse_SDK	StyledString.java	118	165
15	16860613	189	204	eclipse_SDK	StyledString.java	189	204
16	19567163	1	10	eclipse_SDK	AbstractDecoratedTextEditor.java	1351	1360
17	2513183	168	181	eclipse_SDK	GenerateToStringAction.java	175	188
18	6092014	1	10	eclipse_SDK	ViewParameterValues.java	8	17
19	8065120	1	11	eclipse_SDK	FormToolkit.java	287	297
20	968656	1	11	eclipse_SDK	MethodStubsSelectionButtonGroup.java	207	217
21	11861598	1	18	eclipse_SDK	WizardDialog.java	377	394
22	10289462	4	23	eclipse_SDK	JspRuntimeLibrary.java	260	279
23	16928749	50	66	hadoop-1	TextOutputFormat.java	46	63
24	21702608	1	36	hadoop-1	DBCountPageView.java	275	306
25	801987	1	18	hadoop-1	StringUtils.java	41	57
26	16928749	68	97	hadoop-1	TextOutputFormat.java	73	99
27	18647984	48	63	hadoop-1	SequenceFileRecordReader.java	36	50
28	22315734	10	21	hadoop-1	DeserializerComparator.java	16	26
29	14845581	1	10	hadoop-1	JobSubmissionFiles.java	46	55
30	16180910	1	14	hadoop-1	LineRecordReader.java	47	60
31	21702608	15	35	hadoop-1	DBCountPageView.java	289	309
32	23520731	1	75	hibernate-release-4	SchemaUpdate.java	115	190
33	23967852	1	20	hibernate-release-4	SQLServer2005LimitHandler.java	43	62
34	24924255	1	32	hibernate-release-4	Example.java	218	247
35	16930707	41	59	hibernate-release-4	RegisterUserEventListenersTest.java	40	53
36	609430	1	17	hibernate-release-4	DefaultLoadEventListener.java	277	293
37	10274267	1	12	hibernate-release-4	flush_and_clear_session.java	1	12
38	14330686	1	12	hibernate-release-4	flush_and_clear_session.java	1	12
39	14582029	1	12	hibernate-release-4	flush_and_clear_session.java	1	12
40	15168494	1	29	hibernate-release-4	ConnectionProviderInitiator.java	65	93

Table B.2: 100 outdated cloned code snippets on Stack Overflow and their associated original projects (cont.)

No.	Stack Overflow Post	Start	End	Qualitas Project	File	Start	End
41	19298607	1	10	hibernate-release-4	Oracle9iDialect.java	23	32
42	20458485	1	12	hibernate-release-4	flush_and_clear_session.java	1	12
43	21777900	1	12	hibernate-release-4	flush_and_clear_session.java	1	12
44	23520731	55	72	hibernate-release-4	SchemaUpdate.java	170	187
45	23974103	1	34	hibernate-release-4	BulkOperationCleanupAction.java	124	157
46	2398335	1	16	hibernate-release-4	using_scroll.java	1	16
47	2761630	1	11	hibernate-release-4	using_a_StatelessSession.java	1	11
48	3275733	1	12	hibernate-release-4	flush_and_clear_session.java	1	12
49	3788516	1	12	hibernate-release-4	flush_and_clear_session.java	1	12
50	5713930	1	16	hibernate-release-4	using_scroll.java	1	16
51	8037824	1	20	jasperreports-3	JRVerifier.java	1221	1240
52	8037824	22	38	jasperreports-3	JRVerifier.java	982	998
53	21734562	24	48	jboss-5	HTTPBasicServerAuthModule.java	63	87
54	21734562	58	68	jboss-5	HTTPBasicServerAuthModule.java	97	107
55	11368428	1	27	jfreechart-1	BarChartDemo1.java	29	55
56	12936580	63	197	jfreechart-1	AbstractXYItemRenderer.java	443	578
57	21998949	10	89	jfreechart-1	SpiderWebPlot.java	502	578
58	25651812	43	83	jfreechart-1	BarChartDemo1.java	56	87
59	12936580	83	94	jfreechart-1	AbstractXYItemRenderer.java	466	477
60	12936580	102	113	jfreechart-1	AbstractXYItemRenderer.java	484	495
61	16058183	1	13	jfreechart-1	KeyToGroupMap.java	18	30
62	21998949	31	45	jfreechart-1	SpiderWebPlot.java	522	536
63	6722760	38	56	jgraph-latest-bsd-src	MyCellView.java	53	73
64	15889119	1	16	jgraph-0	TouchgraphConverter.java	7	22
65	14940863	1	26	jstock-1.0.7c	GoogleMail.java	18	43
66	3629882	1	26	jstock-1.0.7c	Utils.java	1484	1502
67	6025026	212	233	jung2-2.0.1	ShortestPathDemo.java	158	179
68	10042010	30	41	jung2-2.0_1	TreeLayout.java	85	96
69	10042010	42	55	jung2-2.0_1	TreeLayout.java	97	110
70	24330611	197	213	jung2-2.0_1	TreeCollapseDemo.java	142	157
71	24330611	320	333	jung2-2.0_1	VertexCollapseDemoWithLayouts.java	247	258
72	6025026	47	93	jung2-2.0_1	ShortestPathDemo.java	42	82
73	6025026	137	153	jung2-2.0_1	ShortestPathDemo.java	121	137
74	24330611	162	177	jung2-2.0_1	L2RTreeLayoutDemo.java	102	117
75	24330611	321	347	jung2-2.0_1	TreeCollapseDemo.java	248	274
76	8802082	1	20	junit-4	ExpectException.java	12	31
77	23586872	1	20	junit-4	Assert.java	33	52
78	17697173	1	19	lucene-4.3.0	SlowSynonymFilterFactory.java	38	52
79	18970685	1	10	lucene-4.3.0	FSDirectory.java	35	44
80	12593810	1	12	poi-3.6-20091214	ExtractorFactory.java	49	60
81	18201985	1	26	poi-3.6-20091214	CalendarDemo.java	11	36
82	10924700	1	18	spring-framework-3.0.5	Jaxb2Marshaller.java	253	270
83	6149818	1	20	spring-framework-3.0.5	DefaultPropertiesPersister.java	67	86
84	20913543	1	15	spring-framework-3.0.5	AutowireUtils.java	30	43
85	249149	1	11	spring-framework-3.0.5	CciTemplate.java	193	203
86	7099864	13	27	spring-framework-3.0.5	OptionTag.java	84	98
87	9003314	70	83	spring-framework-3.0.5	WebDataBinder.java	95	108
88	18623736	1	39	spring-framework-3.0.5	CustomCollectionEditor.java	33	71
89	20421869	22	51	spring-framework-3.0.5	ClassPathScanningCandidateComponentProvider.java	90	119
90	20996373	1	15	spring-framework-3.0.5	DelegatingServletInputStream.java	6	20
91	22865824	67	80	spring-framework-3.0.5	JavaMailSenderImpl.java	186	199
92	3751463	1	11	spring-framework-3.0.5	ScheduledTasksBeanDefinitionParser.java	42	52
93	3758110	5	19	spring-framework-3.0.5	DefaultAnnotationHandlerMapping.java	78	92
94	4781746	1	13	spring-framework-3.0.5	DispatcherServlet.java	91	103
95	5660519	1	10	spring-framework-3.0.5	AnnotationMethodHandlerExceptionResolver.java	224	233
96	14019840	1	18	struts2-2.2.1-all	DefaultActionMapper.java	128	145
97	15110171	23	40	struts2-2.2.1-all	StringLengthFieldValidator.java	25	42
98	15131432	5	22	struts2-2.2.1-all	FreemarkerManager.java	144	162
99	15131432	23	42	struts2-2.2.1-all	FreemarkerManager.java	163	177
100	15110171	2	24	struts2-2.2.1-all	StringLengthFieldValidator.java	4	26

Appendix C

Chapter 5: The OCD Framework

C.1 The complete list of the optimal configurations

Table C.1: The complete list of optimal configurations for Krakatau

Tool	Settings	Granularity	Threshold
ccfx	b=5, t=8	T	50
	b=5, t=11	T	17
deckard	mintoken=30, stride=1, similarity=0.95	L, T, C	29,29,34
	mintoken=30, stride=1, similarity=1.00	L, T, C	10,12,15
	mintoken=30, stride=2, similarity=0.90	T	54
	mintoken=30, stride=2, similarity=0.95	L, T, C	22,28,32
	mintoken=30, stride=inf, similarity=0.95	L, T, C	29,29,34
	mintoken=30, stride=inf, similarity=1.00	L, T, C	10,12,15
	mintoken=50, stride=1, similarity=0.95	L, T, C	23,29,31
	mintoken=50, stride=2, similarity=0.95	L, T, C	21,18,23
	mintoken=50, stride=inf, similarity=0.95	L, T, C	23,28,31
simian	threshold=3,4, ignoreidentifiers	T	17
	threshold=3, ignoreidentifiers	C	
	threshold=5, ignoreidentifiers	L	12
	threshold=5, ignoreidentifiers	T	13
simjava	r=18,19		17
	r=20		16
	r=26,27		11
	r=28		9
	r=default		27
	r=4		33
simtext	r=5		31
	ma=LZMA, mx=7,9		54
icd	ma=LZMA2, mx=7,9		

C.2 The OCD Framework User’s Guide

1. Download the OCD framework from <http://crest.cs.ucl.ac.uk/resources/cloplag> to your designated directory and extract it.
2. The framework folder should contain the following structure.

```
.
|---- programs
|---- tests
|---- tests_no_krakatau
|---- tests_no_procyon
|---- scripts
|---- results
|    |---- thresholds
|---- results_no_krakatau
|    |---- thresholds
|---- results_no_procyon
|    |---- thresholds
```

The **programs** folder is where you put the code similarity tools being evaluated. The three folders starting with **tests** contain the OCD data set without compilation/decompilation (**tests**), the OCD data set with compilation/decompilation by Krakatau (**tests_no_krakatau**), and the OCD data set with compilation/decompilation by Procyon (**tests_no_procyon**). The **scripts** folder contains tool running scripts and other scripts required for reading the similarity report and compute the error measures. The folders with names starting with **results** contain similarity reports generated by running a tool on each respective OCD data set. The **thresholds** folder inside each **results** folder contains scripts to compute F1 scores, precision-at-n, ARP, and MAP.

3. Create a running script for your tool. You can adapt from an example script of Linux’s diff tool shown below (Figure C.1). The actual script can be found in the **scripts** folder. You are supposed to modify the content of the **m_compare** function to call your tool and derive a similarity value of two given files **\$1** and **\$2**. Please make sure the tool returns a similarity value from 0 to 100.

```

1 #!/bin/sh
2 # Code similarity tool: Linux's diff
3 # Authors: Jens Krinke and Chaiyong Ragkhitwetsagul
4 DETECTOR=diff
5 # Do the comparison between two tests.
6 m_compare() {
7   # call diff to generate the diff file
8   diff -i -E -b -w -B -e $1/*.java $2/*.java > ../$RESULTS/files.diff
9   # get the file size of files.diff
10  diffSize='cat ../$RESULTS/files.diff | wc -c'
11  fileSize='cat $2/*.java | wc -c'
12  if [ $diffSize -lt $fileSize ] ; then
13    echo "100 - ((${diffSize} * 100 / ${fileSize})" | bc
14  else
15    echo "100"
16  fi
17  rm -rf ../$RESULTS/files.diff
18 }
19 # Set the directory for the results based on the current test directory.
20 RESULTS='basename $PWD | sed -e s:tests:results: | sed -e s:soco:results_soco:'
21 if [ `echo "1\t2"" = "1\t2` ]; then
22   ECHO="-e"
23 else
24   ECHO=""
25 fi
26 # Create the table header.
27 LINE="_"
28 for p in *; do
29   for i in $p/[0-9A-Za-z]*; do # $p/test_*
30     LINE="$LINE, $i"
31   done
32 done
33 echo $LINE > ../$RESULTS/$DETECTOR.csv
34 count=1
35 # Do the pairwise comparisons.
36 for p in *; do
37   for i in $p/[0-9A-Za-z]*; do # $p/test_*
38     LINE="$i"
39     for q in *; do
40       for j in $q/[0-9A-Za-z]*; do # $q/test_*
41         sim=$(m_compare $i $j)
42         echo $ECHO "$count: diff $sim: $i $j"
43         count=$((count+1))
44         LINE="$LINE, $sim"
45     done
46   done
47   echo $LINE >> ../$RESULTS/$DETECTOR.csv
48 done
49 done

```

Figure C.1: An example of a tool running script (`compare_diff.sh`)

4. Change directory to `tests` folder and run the script. The following example runs the script of the Linux's diff tool.

```

$cd tests
../../scripts/compare_diff.sh

```

5. 10,000 pairwise comparisons will be performed by the script (you will see the file names being compared and the similarity score printed to the screen). Once the execution is complete, you will find a similarity report (`diff.csv`) in the `results` folder.

```
$cd ../results  
$ls  
diff.csv    diff.info    thresholds/
```

6. To compute the error measure, both pair-based and query-based ones, change directory to the **thresholds** folder and execute the scripts. For example, to compute precision, recall, accuracy, and F1 scores, run the **all_f1.sh** script as shown below.

```
$cd thresholds  
$./all_f1.sh  
file,T,fp,fn,prec,rec,acc,f1  
diff,8,816,626,8374,184,0.565880721220527,0.816,0.919,0.6683046683046682
```

Appendix D

Chapter 7: Siamese – A User’s Guide

1. Siamese executable (JAR file) can be downloaded from <https://siamesetool.github.io/siamese/>.
2. Please make sure you have Java 8 installed on your machine.
3. To execute Siamese, unzip the file and follow the steps below:

```
$cd siamese  
$./elasticsearch-2.2.0/bin/elasticsearch -d  
$java -jar siamese-0.0.5-SNAPSHOT.jar
```

4. Then, the usage and examples of how to run Siamese will be shown on screen.

```
usage: (v 0.5) $java -jar siamese.jar -cf <config file> [-i input] [-o output] [-c command] [-h help]  
Example: java -jar siamese.jar -cf config.properties  
Example: java -jar siamese.jar -cf config.properties -i /my/input/dir -o /my/output/dir -c index  
-c,--command <arg>      [optional] command to execute [index, search].  
                           This will override the configuration file.  
-cf,--configFile <arg>   [* required *] a configuration file  
-h,--help                <optional> print help  
-i,--inputFolder <arg>   [optional] location of the input files (for  
                           index or query). This will override the  
                           configuration file.  
-o,--outputFolder <arg>  [optional] location of the search result file.  
                           This will override the configuration file.
```

Appendix E

Chapter 8: Detection of GitHub Projects' Licenses

Table E.1: A list of GitHub project's license keywords used by Siamese

License	Pattern
MIT	“THE SOFTWARE IS PROVIDED ‘‘AS IS’’, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED”
GPL-2.0	“GNU GENERAL PUBLIC LICENSE” AND “Version 2”
GPL-3.0	“GNU GENERAL PUBLIC LICENSE” AND “Version 3”
Apache-2.0	“Apache License, Version 2.0”
BSD	“BSD” “ Redistributions of source code” “ Redistributions in binary form”
BSD-2-clause	“Neither the name”
BSD-3-clause	“NO EXPRESS OR IMPLIED LICENSES TO ANY PARTY’S PATENT RIGHTS ARE GRANTED BY THIS LICENSE”
Unlicense	“ http://unlicense.org ”
LGPL	“GNU LESSER GENERAL PUBLIC LICENSE”
LGPL-2.1	“Version 2.1”
LGPL-3.0	“Version 3”
AGPL-3.0	“GNU AFFERO GENERAL PUBLIC LICENSE” AND “Version 3”
MPL-2.0	“Mozilla Public License Version 2.0”
OSL-3.0	“Open Software License version 3.0”
AFL-3.0	““Academic Free License version 3.0””
Artistic-2.0	“Artistic License 2.0””
CC0-1.0	“CC0 1.0””
CC-BY-4.0	“Creative Commons Attribution 4.0””
CC-BY-SA-4.0	“Attribution-NonCommercial-ShareAlike 4.0””
WTFPL	“DO WHAT THE FUCK YOU WANT TO””
ECL-2.0	“Educational Community License, Version 2.0”” OR “ECL-2.0””
EPL-1.0	“Eclipse Public License - v 1.0””
EUPL-1.1	“EUPL V.1.1””
ISC	“ISC License””
LPPL-1.3c	“LPPL Version 1.3c””
MS-PL	“Microsoft Public License”” OR ““(MS-PL)””
POSTGRESQL	“PostgreSQL License””
OFL-1.1	“SIL Open Font License”” AND “version 1.1””
NCSA	“NCSA Open Source License””
ZLIB	Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software””
Unknown	The rest that are not matched with the above patterns.

Bibliography

- R. Abdalkareem, E. Shihab, and J. Rilling. On code reuse from StackOverflow: An exploratory study on Android apps. *Information and Software Technology*, 88: 148–158, Aug 2017.
- Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You get where you’re looking for: The impact of information sources on code security. In *Proceedings of the IEEE Symposium on Security and Privacy (SP ’16)*, pages 289–305, 2016.
- Adjust. Birth, life and death of an app. a look at the Apple App Store in July 2014. https://www.adjust.com/assets/downloads/AppleAppStore_Report2014.pdf, 2014. Accessed: 23-Apr-2015.
- A. Ahtiainen, S. Surakka, and M. Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research (Baltic Sea ’06)*, pages 141–142, Koli, Joensuu, Finland, 2006. ACM.
- A. Aiken. MOSS (measure of software similarity). <https://theory.stanford.edu/~aiken/moss/>, 2015. Accessed: 24-Apr-2015.
- M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM ’12)*, pages 295–304, 2012.

- L. An, O. Mlouki, F. Khomh, and G. Antoniol. Stack Overflow: A code laundering platform? In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER '17)*, 2017.
- Apache Software Foundation. TF-IDF similarity. https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html, 2012. Online; access =20-March-2017.
- A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1-4):315–336, 1987.
- Aragon Consulting Group, Inc. Krugle. <http://krugle.com>, 2018. Online; access 23-April-2018.
- R. ASE group. Ijadataset 2.0. <https://sites.google.com/site/asegsecold/projects/seclone>, 2018. Online; access 13-March-2018.
- L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07)*, pages 81–90, Los Alamitos, California, USA, 2007. IEEE.
- S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, page 157, 2010.
- S. Baltes, R. Kiefer, and S. Diehl. Attribution required: Stack Overflow code snippets in GitHub projects. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C'17)*, pages 161–163, 2017.
- E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE '14)*, pages 306–317. ACM, 2014.

- V. Bauer, T. Volke, and E. Jurgens. A novel approach to detect unintentional re-implementations. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*, pages 491–495, 2014.
- V. Bauer, T. Volke, and S. Eder. Combining clone detection and latent semantic indexing to detect re-implementations. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, pages 23–29, 2016.
- I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, volume 98, pages 368–377, 1998.
- S. Bazrafshan and R. Koschke. An empirical study of clone removals. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '13)*, pages 50–59, 2013.
- N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP '11)*, pages 2–26, 2011.
- S. M. Beitzel, E. C. Jensen, and O. Frieder. *Average R-Precision*, pages 195–195. Springer US, 2009.
- S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.
- H. L. Berghel and D. L. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 19(8):65, 1984.
- L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, pages 39–48, A Curuna, Spain, 2000.

- B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*, 2011.
- BlackDuck. OpenHub. <http://code.openhub.net>, 2016. Online; access 18-May-2016.
- B. D. BlackDuck CORSI. 2017 BlackDuck open source security and risk analysis. Technical report, Black Duck, 2017.
- A. Bosu, C. S. Corley, D. Heaton, D. Chatterji, J. C. Carver, and N. A. Kraft. Building reputation in StackOverflow: An empirical investigation. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, pages 89–92, 2013.
- G. E. Box, J. S. Hunter, and W. G. Hunter. *Statistics for Experimenters*. John Wiley and Sons, 1978.
- Boyster, Ben. Searchcode. <https://searchcode.com>, 2018. Online; access 23-April-2018.
- P. T. Breuer and J. P. Bowen. Decompilation: the enumeration of types and grammars. *Transactions on Programming Languages and Systems*, 16(5):1613–1647, 1994.
- R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *Proceedings of the 10th Working Conference on Source Code Analysis and Manipulation (SCAM '10)*, pages 77–86, 2010.
- C. Brown and S. Thompson. Clone detection and elimination for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '10)*, page 111, 2010.

- E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM '02)*, pages 36–43, 2002.
- S. Burrows, S. M. M. Tahaghoghi, and J. Zobel. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, 37(2):151–175, 2007.
- Business Software Alliance (BSA). Eight annual BSA global software: 2010 piracy study. Technical report, Business Software Alliance (BSA), 2011.
- S. Carter, R. Frank, and D. Tansley. Clone detection in telecommunications software systems: A neural net approach. In *Proceedings of the International Workshop on Application of Neural Networks to Telecommunications*, pages 273–287, 1993.
- A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-checking oracles from intrinsic software redundancy. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, 2014.
- A. Carzaniga, A. Mattavelli, and M. Pezzè. Measuring software redundancy. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Florence, Italy, 2015. ACM.
- M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC '09)*, pages 178–187, 2009.
- M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, pages 1040–1074, 2013.
- D. K. Chae, J. Ha, S. W. Kim, B. Kang, and E. G. Im. Software plagiarism detection: A graph-based approach. In *Proceedings of the 22nd International Conference*

on information & knowledge management (CIKM '13), pages 1577–1580. ACM, 2013.

D. Chatterji, J. C. Carver, and N. A. Kraft. Code clones and developer behavior: results of two surveys of the clone research community. *Empirical Software Engineering*, 21(4):1476–1508, Aug 2016.

K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, Hyderabad, India, 2014. ACM.

X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, 2004.

W. T. Cheung, S. Ryu, and S. Kim. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering*, pages 517–564, 2015.

M. Choetkertikul, D. Avery, H. K. Dam, T. Tran, and A. Ghose. Who will answer my question on Stack Overflow? In *Proceedings of the 24th Australasian Software Engineering Conference (ASWEC '15)*, pages 155–164, 2015.

S. Chow, S. Chow, Y. Gu, Y. Gu, H. Johnson, H. Johnson, V. a. Zakharov, and V. a. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proceedings of the 4th International Conference on Information Security (ISC '01)*, pages 144–155, 2001.

C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *Transactions on Information Theory*, 51(4):1523–1545, 2005.

- R. Cilibrasi, A. L. Cruz, S. de Rooij, and M. Keijzer. Comlearn. <http://comlearn.org/index.html>, 2015. Accessed: 2016-02-14.
- P. Clough. Plagiarism in natural and programming languages: An overview of current tools and technologies. Technical Report July, University of Sheffield, Sheffield, UK, 2000.
- A. Cohen. FuzzyWuzzy: Fuzzy string matching in Python. <http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/>, 2011. Accessed: 2016-02-14.
- C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *Technical Report 148, Dept. of Computer Science, The University of Auckland*, 148, 1997.
- C. Collberg, G. Myles, and A. Huntwork. Sandmark – A tool for software protection research. *Security and Privacy*, 1(4):40–49, 2003.
- C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '04)*, volume 39, pages 107–118, Washington DC, USA, 2004.
- C. S. Collberg, C. Thomborson, and S. Member. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *Computer*, 28(8):735–746, 2002.
- Computer Associates Int'l, Inc. v. Altai, Inc. Computer Associates Int'l, Inc. v. Altai, Inc. 982 F.2d 693 (2d Cir. 1992). <http://cyber.law.harvard.edu/people/tfisher/IP/1992Altai.pdf>, 1992. Accessed: 22-Apr-2015.
- Compuware Corp. v. International Business Machines (IBM). Compuware Corp. v. International Business Machines, 259 F. Supp. 2d 597 (E.D. Mich. 2002). <http://law.justia.com/cases/federal/appellate-courts/cafc/13-1021/13-1021-2014-05-09.html>, 2002. Accessed: 2-June-2015.

- J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
- J. R. Cordy and C. K. Roy. The NiCad clone detector. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC '11)*, pages 219–220. IEEE, 2011.
- G. Cosma. *An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis*. PhD thesis, University of Warwick, 2008.
- G. Cosma and M. Joy. Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, 51(2):195–200, 2008.
- G. Cosma and M. Joy. An approach to source-code plagiarism detection and investigation using Latent Semantic Analysis. *IEEE Transactions on Computers*, 61(3):379–394, 2012.
- N. Craswell. *Encyclopedia of Database Systems*. Springer US, 2009.
- J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS '12)*, pages 37–54, Pisa, Italy, 2012. Springer.
- J. Crussell, C. Gibler, and H. Chen. AnDarwin: Scalable detection of semantically similar android applications. In *Proceddings of the European Symposium on Research in Computer Security (ESORICS '13)*, pages 182–199, Egham, UK, 2013. Springer.
- C. Daniela, P. Navrat, B. Kovacova, and P. Humay. The issue of (software) plagiarism: A student view. *IEEE Transactions on Education*, 55(1):22–28, 2012.
- N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3-4), 1995.

- J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software Bertillonage: Determining the provenance of software development artifacts. *Empirical Software Engineering*, 18:1195–1237, 2013.
- I. J. Davis and M. W. Godfrey. From Where It Came: Detecting Source Code Clones by Analyzing Assembler. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pages 242–246, 2010.
- T. Debatty. Java string similarity. <https://github.com/tdebatty/java-string-similarity>, 2018. Accessed: 2018-06-10.
- F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 13th International Conference on Software Engineering (ICSE '08)*, pages 603–612, 2008.
- S. Designs. SD’s source code obfuscators. <http://www.semdesigns.com/Products/Obfuscators/>, 2015. Accessed: 02-June-2015.
- A. Desnos and G. Gueguen. Android: From reversing to decompilation. *Black Hat Abu Dhabi*, pages 1–24, 2011.
- M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*, pages 145–154, Cape Town, South Africa, 2010. ACM.
- T. Diamantopoulos and A. L. Symeonidis. Employing source code information to improve question-answering in Stack Overflow. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*, pages 454–457, 2015.
- J. Donaldson, A. Lancaster, and P. Sposato. A plagiarism detection system. *ACM SIGCSE Bulletin*, pages 21–25, 1981.

- E. Duala-Ekoko and M. P. Robillard. Clonetracker: Tool support for code clone management. In *Proceedings of the 13th international conference on Software engineering (ICSE '08)*, 2008.
- S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, pages 109–118, Leicester, UK, 1999. IEEE.
- Z. Duric and D. Gasevic. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, Mar. 2013.
- T. I. Economy. The power of a mismanaged intangible. https://intangibleeconomy.wordpress.com/2009/08/05/skype_in_danger_of_shutdown_over_p2p_licensing_say/, 2009. Accessed: 1-March-2017.
- Elasticsearch BV. Elasticsearch. <https://www.elastic.co/products/elasticsearch>, 2016. Online; access 25-Jun-2016.
- J. Faidhi and S. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19, 1987.
- Fair Use. Intellectual property for CS students: Copyrights – fair use. <https://www.cs.duke.edu/courses/cps182s/fall02/cscopyright/Copyright/Copyright-Fairuse.htm>, 2017. Accessed: 2018-06-27.
- R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, Dec 2008.
- F. Fischer, K. Bottinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow considered harmful? the impact of copy&paste on Android application security. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP '17)*, pages 121–136, 2017.
- R. Fisher. *The Design of Experiments*. Oliver and Boyd, 1935.

- E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello. Detection of source code re-use. <http://users.dsic.upv.es/grupos/nle/soco/>, 2014. Accessed: 2016-02-14.
- M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- M. Fowler. Catalog of refactorings. <https://refactoring.com/catalog/>, 2013. Accessed: 2017-05-28.
- M. Funaro, D. Braga, A. Campi, and C. Ghezzi. A hybrid approach (syntactic and textual) to clone detection. In *Proceedings of the 4th International Workshop on Software Clones (IWSC '10)*, pages 79–80, New York, New York, USA, 2010. ACM.
- M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 13th International Conference on Software Engineering (ICSE '08)*, page 321, 2008.
- R. E. Gallardo-Valencia and S. E. Sim. Internet-scale code search. *Proceedings of the ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE '09)*, pages 49–52, 2009.
- D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR '09)*, pages 81–90. IEEE, 2009.
- D. M. German, Y. Manabe, and K. Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE '10)*, page 437, 2010.
- C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. AdRob: Examining the landscape and impact of android application plagiarism. In *Proceedings of*

the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys '13), page 431, Taipei, Taiwan, 2013. ACM.

- D. Gitchell and N. Tran. Sim: A utility for detecting similarity in computer programs. In *The proceedings of the 30th SIGCSE technical symposium on Computer science education (SIGCSE '99)*, volume 31, pages 266–270, New Orleans, Louisiana, USA, 1999.
- R. Gobeille. The FOSSology project. In *Proceedings of the 2008 international workshop on Mining software repositories (MSR '08)*, page 47. ACM, 2008.
- N. Göde and J. Harder. Clone stability. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR '11)*, pages 65–74, 2011.
- N. Göde and R. Koschke. Incremental clone detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR '09)*, pages 219–228, Kaiserslautern, Germany, 2009. IEEE.
- M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '10)*, pages 475–484, 2010.
- S. Grier. A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bulletin*, 13(1):15–20, 1981.
- G. Gross. Open-source legal group strikes again on BusyBox, suing Verizon. <http://www.computerworld.com/article/2537947/open-source-tools/open-source-legal-group-strikes-again-on-busybox--suing-verizon.html>, 2007. Accessed: 20-May-2015.
- R. Grosse. Krakatau bytecode tools. <https://github.com/Storyyeller/Krakatau>, 2016. Accessed: 2016-02-14.

- X. Gu, H. Zhang, and S. Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pages 933–944, 2018.
- Guard Square. ProGuard: Bytecode obfuscation tool. <http://proguard.sourceforge.net>, 2015. Accessed: 02-June-2015.
- J. Hage, P. Rademaker, and N. van Vugt. A comparison of plagiarism detection tools. Technical Report UU-CS-2010-015, Department of Information and Computing Sciences Utrecht University, Utrecht, The Netherlands, 2010.
- M. H. Halstead. *Elements of Software Science*. Amsterdam: Elsevier North-Holland, Inc., 1977.
- M. A. Hamilton and T. Sabety. Computer science concepts in copyright cases: The path to a coherent law. *Harvard Journal of Law & Technology*, 10:240–280, 1997.
- S. Harris. Simian - Similarity analyser. <http://www.harukizaemon.com/simian/>, 2003. Accessed: 4-May-2015.
- P. Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978.
- A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Proceeding of the 8th working conference on Mining software repositories (MSR '11)*, page 63, 2011.
- A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 837–847, 2012.
- D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977.

- B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *Proceedings of the International Conference on Software Maintenance (ICSM '10)*, pages 1–9. IEEE, 2010.
- J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.
- K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe. Where does this code come from and where does it go? — Integrated code history tracker for open source systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 331–341. IEEE, 2012.
- T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue. Source file set search for clone-and-own reuse analysis. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR '17)*, pages 257–268, 2017.
- K. Jalbert and J. S. Bradbury. Using clone detection to identify bugs in concurrent software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '10)*, pages 1–5, 2010.
- S. Jeong. Federal circuit sends Oracle v. Google back for third trial. <https://www.theverge.com/2018/3/27/17169064/federal-circuit-oracle-v-google-third-trial-java-android>, 2018. Accessed: 27-May-2018.
- L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 96–105, Minneapolis, Minnesota, USA, 2007a. IEEE.
- L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07)*, 2007b.

- M. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.
- M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3), 2005.
- E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering (ICSE'09)*, pages 485–495, 2009.
- E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*, pages 78–87. IEEE, 2011.
- T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilingual token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code : A case study. In *Proceedings of the Evolution of Large-scale Industrial Software Evolution (ELISA '03)*, pages 67–78, Amsterdam, The Netherlands, 2003.
- C. Kapser and M. W. Godfrey. “Cloning considered harmful” considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 19–28, Benevento, Italy, 2006. IEEE.
- C. J. Kapser and M. W. Godfrey. “Cloning considered harmful” considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.
- S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A tool for automatic code clone detection in the IDE. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE '09)*, pages 313–314, 2009.

- D. Kawrykow and M. P. Robillard. Improving API usage through automatic detection of redundant code. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, pages 111–122, 2009.
- Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering (ASE'15)*, pages 295–306, 2015.
- I. Keivanloo, J. Rilling, and P. Charland. Internet-scale real-time code clone search via multi-level indexing. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11)*, pages 23–27. IEEE, 2011.
- I. Keivanloo, C. Forbes, and J. Rilling. Similarity search plug-in: Clone detection meets Internet-scale code search. In *Proceedings of the 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE '12)*, pages 21–22. IEEE, 2012.
- I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the International Conference on Software Engineering (ICSE '14)*, pages 664–675, 2014.
- I. Keivanloo, F. Zhang, and Y. Zou. Threshold-free code clone detection for a large-scale heterogeneous java repository. In *SANER '15*, pages 201–210, 2015.
- H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory comparison-based clone detector. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 301–310, Waikiki, Honolulu, HI, USA, 2011.
- K. Kim, D. Kim, T. F. Bissyande, E. Choi, L. Li, J. Klein, and Y. L. Traon. FaCoY – A code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018.
- M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held*

- jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '13)*, page 187, 2005.
- B. A. Kitchenham and S. L. Pfleeger. Principles of survey research part 2: Designing a survey sample size experimental designs. *Software Engineering Notes*, 27(1):18–20, 2002.
- D. E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.
- R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS '01)*, volume 2126, pages 40–56, Paris, France, 2001. Springer.
- O. Kononenko, C. Zhang, and M. W. Godfrey. Compiling clones: What happens? In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 481–485, 2014.
- R. Koschke. Survey of research on software clones. *Duplication, Redundancy, and Similarity in Software - Dagstuhl Seminar #06301*, page 24, 2007.
- R. Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 26(8):747–769, Aug 2014.
- R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (RE '06)*, pages 253–262, 2006.
- J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE '01)*, pages 301–309. IEEE, 2001.
- J. Krinke. Is cloned code more stable than non-cloned code? In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM '08)*, pages 57–66, 2008.

- J. Krinke, N. Gold, Y. Jia, and D. Binkley. Cloning and copying between GNOME projects. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '10)*, pages 98–101, 2010.
- G. Kumaran and V. R. Carvalho. Reducing long queries using query quality predictors. In *Proceedings of the International Conference on Research and Development in Information Retrieval (SIGIR'09)*, page 564, 2009.
- T. Lavoie, M. Eilers-smith, and E. Merlo. Challenging cloning related problems with GPU-based algorithms. In *Proceedings of the 4th International Workshop on Software Clones (IWSC '10)*, pages 25–32, Cape Town, South Africa, 2010. ACM.
- M. Lee. Free software foundation files suit against Cisco for GPL violations. <http://www.fsf.org/news/2008-12-cisco-suit>, 2008. Accessed: 20-May-2015.
- M. W. Lee, J. W. Roh, S. W. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the 18th International Symposium on Foundations of software engineering (FSE '10)*, page 167. ACM, 2010.
- C. Li, B. Wang, and X. Yang. VGRAM : Improving performance of approximate queries on string collections using variable-length grams. In *International Conference on Very Large Data Bases (VLDB'07)*, pages 303–314, 2007.
- H. Li and S. Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09)*, page 169, 2008.
- H. Li and S. Thompson. Incremental clone detection and elimination for Erlang programs. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE '11)*, pages 356–370, 2011.
- L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. CC Learner: A deep learning-

- based clone detection approach. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'17)*, pages 249–260, 2017.
- M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 3 edition, 2008.
- Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.
- H. I. Lim, H. Park, S. Choi, and T. Han. A method for detecting the theft of java programs through analysis of the control flow information. *Information and Software Technology*, 51(9):1338–1350, 2009.
- Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting differences across multiple instances of code clones. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pages 164–174, 2014.
- E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336, April 2009.
- C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06)*, page 872, Philadelphia, PA, USA, 2006. ACM.
- S. Livieri, D. M. German, and K. Inoue. A needle in the stack: Efficient clone detection for huge collections of source code. Technical report, Osaka University, 2010.
- C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. DéjàVu: A map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 1:1–28, Oct 2017.

- L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*, pages 389–400, New York, New York, USA, 2014. ACM.
- M. Madou, L. V. Put, and K. D. Bosschere. Loco: An interactive code (de) obfuscation tool. In *Proceedings of the 2006 Symposium on Partial evaluation and semantics-based program manipulation (PEPM '06)*, pages 140–144, 2006.
- C. D. Manning, P. Raghavan, and H. Schutze. *An Introduction to Information Retrieval*, volume 21. Cambridge University Press, 2009.
- L. Martie, A. V. D. Hoek, and T. Kwak. Understanding the impact of support for iteration on code search. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*, pages 774–785, 2017.
- C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usages. In *Proceedings of the International Conference on Software Engineering (ICSE '11)*, page 111, 2011.
- C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 364–374, Zurich, Switzerland, 2012. IEEE.
- G. Miller. The magic number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, 1956.
- M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider. An empirical study of the impacts of clones in software maintenance. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC '11)*, pages 242–245, 2011.
- M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider. Is cloned code really stable? *Empirical Software Engineering*, 23(2):693–770, Apr 2018.

- B. Morgenstern, K. Frech, A. Dress, and T. Werner. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics (Oxford, England)*, 14(3):290–294, 1998.
- D. Movshovitz-Attias, Y. Movshovitz-Attias, P. Steenkiste, and C. Faloutsos. Analysis of the reputation system and user contributions on a question answering website: StackOverflow. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining (ASONAM '13)*, pages 886–893, 2013.
- H. Murakami, Y. Higo, and S. Kusumoto. A dataset of clone references with gaps. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*, pages 412–415, 2014.
- A. Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). *Programming Languages and Systems*, 1999.
- G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In *Proceedings of the 7th International Conference on Information Security (ISC '04)*, volume 3225, pages 404–415, 2004.
- G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the Symposium on Applied Computing (SAC '05)*, pages 314–318, Santa Fe, New Mexico, 2005. ACM.
- C. Nachenberg. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers*, 30:16, 1996.
- S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM '12)*, pages 25–34, 2012.
- G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the*

International Conference on Compiler Construction (CC '02), pages 213–228, 2002.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE '09)*, pages 440–455, 2009a.

H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.

T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen. Scalable and incremental clone detection for evolving software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '09)*, pages 491–494, 2009b.

M. A. Nishi and K. Damevski. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software*, 137:130–142, Mar 2018.

H. Niu, I. Keivanloo, and Y. Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, 22(1):259–291, Feb 2017.

T. Ohmann and I. Rahal. Efficient clustering-based source code plagiarism detection using PIY. *Knowledge and Information Systems*, 41(1), 2014.

V. Oksanen and M. Kupsu. Open Source License Checker (OSLC). <http://forge.ow2.org/projects/oslcv3>, 2016. Online; access 12-March-2017.

- C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the International Conference on Software Engineering (ICSE '12)*, pages 859–869, 2012.
- Oracle America, Inc. v. Google Inc. Oracle America, Inc. v. Google Inc., No. 3:2010cv03561 - Document 642 (N.D. Cal. 2011). <http://law.justia.com/cases/federal/district-courts/FSupp2/259/597/2362960>, 2011. Accessed: 23-Apr-2015.
- Oracle America, Inc. v. Google Inc. ORACLE AMERICA, INC. v. GOOGLE INC., No. 13-1021 (Fed. Cir. 2012). <http://law.justia.com/cases/federal/appellate-courts/cafc/13-1021/13-1021-2012-12-10.html>, 2012. Accessed: 23-Apr-2015.
- K. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 1976.
- G. U. PARIS research group, ELIS department. Diablo: Code obfuscator. <http://diablo.elis.ugent.be/obfuscation>, 2015. Accessed: 02-June-2015.
- J. W. Park, M. W. Lee, J. W. Roh, S. W. Hwang, and S. Kim. Surfacing code in the dark: an instant clone search approach. *Knowledge and Information Systems*, 41(3):727–759, 2014.
- T. Parr, S. Harwell, and I. Kochurkin. Grammars written for ANTLR v4. <https://github.com/antlr/grammars-v4>, 2017. Accessed: 2017-11-21.
- J. R. Pate, R. Tairas, and N. A. Kraft. Clone evolution: A systematic review. *Journal of software: Evolution and Process*, 25:261–283, 2013.
- I. Pavlov. 7-Zip. <http://www.7-zip.org>, 2016. Accessed: 2016-02-14.
- F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

- S. L. Pfleeger and B. A. Kitchenham. Principles of survey research part 1: Turning lemons into lemonade. *Software Engineering Notes*, 26(6):16, 2001.
- N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE '09)*, pages 276–286, 2009.
- R. Pike and Loki. The Sherlock plagiarism detector. <http://www.cs.usyd.edu.au/~scilect/sherlock/>, 2002. Accessed: 2016-02-14.
- PMD’s Copy/Paste Detector (CPD). PMD’s Copy/Paste Detector (CPD). <https://pmd.github.io/pmd-5.8.1/usage/cpd-usage.html>, 2017. Accessed: 2018-06-27.
- L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack Overflow in the IDE. In *Proceedings of the International Conference on Software Engineering (ICSE '13)*, pages 1295–1298, 2013.
- L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*, pages 102–111, 2014.
- G. Poulter. Python ngram 3.3. <https://pythonhosted.org/ngram/>, 2012. Accessed: 2016-02-14.
- L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.
- T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in Java (does bytecode reveal source?). In *USENIX*, pages 185–198, 1997.

- T. Punter, M. Ciolkowski, B. Freimut, and I. John. Conducting on-line surveys in software engineering. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'03)*, pages 80–88, 2003.
- Python Software Foundation. `difflib` – helpers for computing deltas. <http://docs.python.org/2/library/difflib.html>, 2016. Accessed: 2016-02-14.
- C. Ragkhitwetsagul and J. Krinke. The study’s website: Comparison of code similarity analysers. <http://crest.cs.ucl.ac.uk/resources/cloplag/>, 2017. Accessed: 2017-06-20.
- C. Ragkhitwetsagul, J. Krinke, and D. Clark. Similarity of source code in the presence of pervasive modifications. In *Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation (SCAM’16)*, pages 117 – 126, Raleigh, NC, USA, 2016a. IEEE.
- C. Ragkhitwetsagul, M. Paixao, M. Adham, S. Busari, J. Krinke, and J. H. Drake. Searching for configurations in clone evaluation – A replication study. In *Proceedings of the 8th Symposium on Search-Based Software Engineering (SSBSE’16)*, Raleigh, NC, USA, 2016b. Springer.
- C. Ragkhitwetsagul, J. Krinke, and D. Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, 23(4):2464–2519, Aug 2018a.
- C. Ragkhitwetsagul, J. Krinke, and B. Marnette. A picture is worth a thousand words: Code clone detection based on image similarity. In *Proceedings of the 12th International Workshop on Software Clones (IWSC ’18)*, pages 44–50, 2018b.
- F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR ’10)*, pages 72–81, 2010.
- F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

- A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*, volume 67. Cambridge University Press, 2011.
- D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, Jul 2013.
- B. V. Rompaey and S. Demeyer. Establishing traceability links between unit test cases and units under test. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR '09)*, pages 209–218, 2009.
- C. Rosen and E. Shihab. What are mobile developers asking about? A large scale study using Stack Overflow. *Empirical Software Engineering*, 21(3):1192–1223, 2016.
- C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queen’s University at Kingston Ontario, Canada, 2007.
- C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC '08)*, pages 172–181, Amsterdam, Netherlands, 2008. IEEE.
- C. K. Roy and J. R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*, pages 157–166, 2009a.
- C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 26(12):165–189, 2009b.
- C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

- C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: A case study. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '15)*, pages 191–201, 2015.
- V. Saini, H. Sajnani, J. Kim, and C. Lopes. SourcererCC and SourcererCC-I. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*, pages 597–600. ACM Press, 2016a.
- V. Saini, H. Sajnani, and C. Lopes. Comparing quality metrics for cloned and non cloned Java methods: A large scale empirical study. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME '16)*, pages 256–266, 2016b.
- V. Saini, F. Farmahinifarahan, Y. Lu, P. Baldi, and C. Lopes. Oreo: Detection of clones in the twilight zone. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, 2018.
- H. Sajnai. *Large-Scale Code Clone Detection*. PhD thesis, University of California, Irvine, 2016.
- H. Sajnani, V. Saini, and C. Lopes. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process*, 27(6):402–429, 2015.
- H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the International Conference on Software Engineering (ICSE '16)*, pages 1157–1168, 2016.
- G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.
- SAS Institute, Inc. v. S&H Computer Systems. SAS Institute, Inc. v. S&H COMPUTER SYSTEMS, 605 F. Supp. 816 (M.D. Tenn. 1985).

<http://law.justia.com/cases/federal/district-courts/FSupp/605/816/1658626/>, 1985. Accessed: 22-Apr-2015.

- S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03)*, page 76. ACM, 2003.
- S. Schulze and D. Meyer. On the robustness of clone detection to code obfuscation. In *Proceedings of the 7th International Workshop on Software Clones (IWSC '13)*, pages 62–68, San Francisco, CA, USA, 2013. IEEE.
- G. M. Selim, K. C. Foo, and Y. Zou. Enhancing source-based clone detection using intermediate representation. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pages 227–236, 2010.
- S. E. Sim and R. E. Gallardo-Valencia. *Finding Source Code on the Web for Remix and Reuse*. Springer-Verlag New York, 2013.
- S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology*, 21(1):1–25, Dec 2011.
- M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine*, 25(2):128–131, 2008.
- R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the International Workshop on Software Clones (IWSC '09)*, 2009.
- M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. In *Proceedings of the 16th Conference on information and knowledge management (CIKM '07)*, page 623, 2007.
- C. J. Sprigman. Oracle v. Google. *Communications of the ACM*, 58(5):27–29, 2015.

- K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *Transactions on Software Engineering and Methodology*, 23(3):1–45, 2014.
- M. Strobel. Procyon: Java decompiler. <https://bitbucket.org/mstrobel/procyon/wiki/JavaDecompiler>, 2015. Accessed: 02-June-2015.
- Stunnix. CXX-OBFUS. <http://stunnix.com>, 2015. Accessed: 02-June-2015.
- S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR ’13)*, pages 85–88, 2013.
- J. Svajlenko and C. K. Roy. Evaluating modern clone detection tools. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME ’14)*, pages 321–330. IEEE, 2014.
- J. Svajlenko and C. K. Roy. Evaluating clone detection tools with BigCloneBench. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME’15)*, pages 131–140, 2015.
- J. Svajlenko and C. K. Roy. BigCloneEval: A clone detection tool evaluation framework with BigCloneBench. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME ’16)*, pages 596–600, 2016.
- J. Svajlenko and C. K. Roy. Fast and flexible large-scale clone detection with CloneWorks. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C ’17)*, pages 27–30, 2017.
- J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME ’14)*, pages 476–480, 2014a.

- J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using classical detectors: An exploratory study. *Journal of Software: Evolution and Process*, page 430–464, 2014b.
- H. Tamada, K. Okamoto, and M. Nakamura. Dynamic software birthmarks to detect the theft of Windows applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST '04)*, 2004.
- A. Tamersoy, K. Roundy, and D. H. Chau. Guilt by association: Large scale malware detection by mining. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '14)*, pages 1524–1533, New York, New York, USA, 2014. ACM.
- C. Taube-Schock, R. J. Walker, and I. H. Witten. Can we avoid high coupling? In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP '11)*, pages 204–228, 2011.
- E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC '10)*, pages 336–345, 2010.
- S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014.
- S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, Feb 2010.
- Z. Tian, Q. Zheng, T. Liu, M. Fan, X. Zhang, and Z. Yang. Plagiarism detection for multithreaded software based on thread-aware software birthmarks. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14)*, pages 304–313, 2014.

- C. Treude and M. P. Robillard. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the International Conference on Software Engineering (ICSE '16)*, pages 392–403, 2016.
- N. Tsantalis, D. Mazinanian, and G. P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, 2015.
- N. Tsantalis, D. Mazinanian, and S. Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, pages 60–70, 2017.
- M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the International Conference on Software Engineering (ICSE '15)*, volume 17, pages 403–414, 2015.
- J. Turk and M. Stephens. A python library for doing approximate and phonetic matching of strings. <https://github.com/jamesturk/jellyfish>, 2016. Accessed: 2016-02-14.
- Turnitin. turnitin. <http://turnitin.com>, 2015. Accessed: 19-Apr-2015.
- S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE '05)*, pages 45–56, 2005.
- D. van Bruggen. JavaParser – Process Java code programmatically. <http://javaparser.org>, 2017. Accessed: 2017-11-21.
- A. Vargha and H. D. Delaney. A critique and improvement of the “CL” common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2 (Summer, 2000)):101–132, 2000.
- B. Vasilescu, A. Serebrenik, and M. van den Brand. You can't control the unfamiliar: A study on the relations between aggregation techniques for software

- metrics. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '11)*, pages 313–322, 2011.
- A. Verprauskus. Ohcount. <https://sourceforge.net/projects/ohcount>, 2016. Online; access 12-March-2017.
- N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google play. In *Proceedings of the International conference on Measurement and modeling of computer systems (SIGMETRICS '14)*, pages 221–233, Austin, Texas, USA, 2014.
- C. W. C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '01)*, 2001.
- S. Wang, D. Lo, and L. Jiang. An empirical study on developer interactions in StackOverflow. In *Proceedings of the Symposium on Applied Computing (SAC '13)*, pages 1019–1024, 2013a.
- S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik. EnTagRec: An enhanced tag recommendation system for software information sites. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME '14)*, pages 291–300, 2014.
- T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE '13)*, pages 455–465, Saint Petersburg, Russia, 2013b.
- W. Wang and M. W. Godfrey. Recommending clones for refactoring using design, context, and history. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME '14)*, pages 331–340, 2014.
- X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can I clone this piece

- of code here? In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, page 170, 2012.
- D. Weber-Wulff, K. Köhler, and C. Möller. Collusion detection system test report 2012. <http://plagiat.htw-berlin.de/collusion-test-2012/>, 2012. Accessed: 22-Apr-2015.
- M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.
- G. Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2):140–146, 1990.
- Whelan Associates v. Jaslow Dental Labor. Whelan Associates v. Jaslow Dental Labor., 609 F. Supp. 1307 (E.D. Pa. 1985). <http://law.justia.com/cases/federal/district-courts/FSupp/609/1307/1887159/>, 1985. Accessed: 22-Apr-2015.
- M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*, pages 87–98, 2016.
- F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1 (6):80, 1945.
- M. J. Wise. Detection of similarities in student programs. *ACM SIGCSE Bulletin*, 24(1):268–271, 1992.
- M. J. Wise. String similarity via greedy string tiling and running Karp-Rabin matching. Technical report, Sydney University, 1993.
- M. J. Wise. YAP3: Improved detection of similarities in computer program and other texts. *ACM SIGCSE Bulletin*, 28(1):130–134, 1996.

- P. Xia, M. Matsushita, N. Yoshida, and K. Inoue. Studying reuse of out-dated third-party code. *Information and Media Technologies*, 9(2):155–161, 2014.
- D. Yang, A. Hussain, and C. V. Lopes. From query to usable code: An analysis of Stack Overflow code snippets. In *Proceedings of the Working Conference on Mining Software Repositories (MSR '16)*, pages 391–402, 2016.
- D. Yang, P. Martins, V. Saini, and C. Lopes. Stack Overflow in Github: Any snippets there? In *Proceedings of the International Conference on Mining Software Repositories (MSR '17)*, 2017.
- X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pages 404–415. ACM, 2016.
- F. Zhang, Y. C. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*, pages 111–121, Minneapolis, MN, USA, 2012. ACM.
- F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the Conference on Security and Privacy in Wireless & Mobile networks (WiSec '14)*, pages 25–36, Oxford, United Kingdom, 2014. ACM.
- F. Zhang, H. Niu, I. Keivanloo, and Y. Zou. Expanding queries for code search using semantically related API class-names. *Transactions on Software Engineering*, 2017. doi: 10.1109/TSE.2017.2750682.
- H. Zhang. Exploring regularity in source code: Software science and Zipf's law. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'08)*, pages 101–110, 2008.

- T. Zhang and M. Kim. Automated transplantation and differential testing for clones.
In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, pages 665–676, 2017.
- T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim. Are online code examples reliable? An empirical study of API misuse on Stack Overflow.
In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018.
- Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution.
Proceedings of the 33rd Symposium on Security and Privacy (SP '12), pages 95–109, 2012.
- G. K. Zipf. *Selective Studies and the Principle of Relative Frequency in Language*. Harvard University Press, 1932.