UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Large-Scale Code Clone Detection

Permalink

https://escholarship.org/uc/item/45r2308g

Author

Sajnani, Hitesh

Publication Date

2016

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Large-Scale Code Clone Detection

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Hitesh Sajnani

Dissertation Committee:
Professor Cristina Lopes, Chair
Professor André van der Hoek
Professor James A. Jones

2016

Portions of Chapter 3 © 2016 IEEE
Portions of Chapter 4 © 2016 IEEE
Portions of Chapter 5 © 2015 Wiley & Sons, Inc.
Portions of Chapter 6 © 2016 IEEE
Portions of Chapter 7 © 2014 IEEE
All other materials © 2016 Hitesh Sajnani

DEDICATION

To my parents, sisters, beloved wife, and pramukh swami maharaj.

Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

One of the most important reasons I enjoyed my time at graduate school is because of my relationship with my advisor - Prof. Cristina Lopes (Crista). My deepest gratitude to Crista for these many years of guidance and patience in shaping me into an independent researcher. She gave me the freedom to explore new research ideas, helped me when I faltered, and always motivated me to do my best. I can't think of a better mentor than Crista.

I thank my other committee members: Prof. André van der Hoek and Prof. James Jones for their precious comments on the draft of this dissertation and many thought-provoking conversations during my defense that helped me to make this dissertation better. Many thanks to Prof. Nenad Medivdovic and Prof. Alex Ihler for being part of my advancement committee.

I would like to express my appreciation to my mentors who have given me an industry perspective on research during my internships. In particular, Prof. Chen Li at UCI; Ravindra Naik and Arun Bahulkar at Tata Research; and Rob DeLine, Mike Barnett, Jacek Czerwonka, and Wolfram Schulte at Microsoft Research.

I would like to express my gratitude to the many collaborators in the research community who have helped me to develop my work. My deepest gratitude to Vaibhav Saini, my partner in crime in developing SourcererCC. My thanks to Prof. Chanchal Roy for his constant academic support. Thank you to Jeff Svajlenko and Prof. Roy for their excellent help with the evaluation of SourcererCC. Thank you to Prof. Rainer Koschke for many supportive and constructive comments in the early stages to improve my work.

During the course of my Ph.D., many close friends and family members stood by me through my tough times. I am grateful to all my friends for their support. In particular, Neel, Purvesh and Rupa for being there for me whenever I needed them. I am deeply indebted to my sisters Puja and Neha (Nikky) for being such great friends and taking care of my parents back in India while I was sailing through my graduate studies. I can't thank them enough for all the support and love they have given me. Soumya, my soul-mate, has truly been the most exceptional partner I could wish for. She gave me all the motivation needed to get out of graduate school. She made frequent trips to Irvine to help me through the tough times of dissertation writing and presentation. I really admire her love for me and consider myself blessed to have her in my life. Finally, I would like to thank my wonderful parents, who have made countless sacrifices for the well-being and education of their children. They've sacrificed their life to see me at where I am today. I wish I could learn to love the way they love me. I dedicate this dissertation to them. I thank Lord Swaminarayan for blessing me with such wonderful people in my life.

Funding Acknowledgment. For the majority of the time I was working on this dissertation, I was supported by the National Science Foundation under Grant No. CCF-1218228. Earlier in my graduate career, I was supported by NSF Grant No. CCF-1018374 and through the Department of Informatics at UC Irvine.

Relation to Prior Publications. I am grateful to the following publishers to incorporate some of my previous work into this dissertation. Part of the material in Chapters 3 and 4 is included with the permission of the IEEE and based on work in:

- Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.; Lopes, C., "SourcererCC: Scaling Code Clone Detection to Big-Code," International Conference on Software Engineering (ICSE), May 2016

Part of the material in Chapter 5 is included with the permission of the Wiley and Sons, Inc. and based on work in:

- Sajnani, H.; Saini, V.; Lopes, C., "A Parallel and Efficient Approach to Large Scale Clone Detection," Journal of Software: Evolution and Process (JSEP), June 2015 vol., no. 27, pp. 402-429, doi: 10.1002/smr.1707

Part of the material in Chapter 6 is included with the permission of the IEEE and based on work in:

- Saini, V.; Sajnani, H.; Kim, J.; Lopes, C., "SourcererCC and SourcererCC-I: Tools to Detect Clones in Batch mode and During Software Development," International Conference on Software Engineering (ICSE), May 2016

Part of the material in Chapter 7 is included with the permission of the IEEE and based on work in:

- Sajnani, H.; Saini, V.; Lopes, C., "A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code" Source Code Analysis and Manipulation (SCAM), 2014 14th IEEE Working Conference on, pp.21-30, 28-29 Sept. 2014 doi: 10.1109/S-CAM.2014.12

CURRICULUM VITAE

Hitesh Sajnani

EDUCATION

Doctor of Philosophy in Information and Computer Science **2016**
University of California, Irvine *Irvine, California*

Master of Science in Information and Computer Science **2013**
University of California, Irvine *Irvine, California*

Bachelor of Engineering in Computer Science **2007**
Dharmsinh Desai Institute of Technology *Nadiad, India*

ABSTRACT OF THE DISSERTATION

Large-Scale Code Clone Detection

By

Hitesh Sajnani

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2016

Professor Cristina Lopes, Chair

Clone detection locates exact or similar pieces of code, known as clones, within or between software systems. With the amount of source code increasing steadily, large-scale clone detection has become a necessity. Large code bases and repositories of projects have led to several new use cases of clone detection including mining library candidates, detecting similar mobile applications, detection of license violations, reverse engineering product lines, finding the provenance of a component, and code search.

While several techniques have been proposed for clone detection over many years, accuracy and scalability of clone detection tools and techniques still remains an active area of research. Specifically, there is a marked lack in clone detectors that scale to large systems or repositories, particularly for detecting near-miss clones where significant editing activities may have taken place in the cloned code.

The problem stated above motivates the need for clone detection techniques and tools that satisfy the following requirements: (1) accurate detection of near-miss clones, where minor to significant editing changes occur in the copy/pasted fragments; (2) scalability to hundreds of millions of lines of code and several thousand projects; and (3) minimal dependency on programming languages.

To that effect, this dissertation presents SourcererCC, an accurate, near-miss clone detection tool that scales to hundreds of millions of lines of code (MLOC) on a single standard machine. The core idea of SourcererCC is to build an optimized index of code blocks and compare them using a simple bag-of-tokens strategy, which is very effective in detecting near-miss clones. Coupled with several filtering heuristics that reduce the size of the index, this approach is also very efficient, as it reduces the number of code block comparisons to detect the clones.

This dissertation evaluates scalability, execution time, and accuracy of SourcererCC against four state-of-the-art open-source tools: CCFinderX, Deckard, iClones, and NiCad. To measure scalability, the performance of the tools is evaluated on inter-project software repository IJaDataset-2.0, consisting of $25,000$ projects, containing 3 million files and 250MLOC. To measure precision and recall, two recent benchmarks are used: (1) a benchmark of real clones, BigCloneBench, that spans the four primary clone types and the full spectrum of syntactical similarity in three different languages (Java, C, and C#); and (2) a Mutation/Injection-based framework of thousands of fine-grained artificial clones. The results of these experiments suggest that SourcererCC improves the state-of-the-art in code clone detection by being the most scalable technique known so far, with accuracy at par with the current state-of-the-art tools.

Additionally, this dissertation presents two tools built on top of SourcererCC: (i) SourcererCC-D: a distributed version of SourcererCC that exploits the inherent parallelism present in SourcererCC's approach to scale horizontally on a cluster of commodity machines for large scale code clone detection. Our experiments demonstrate SourcererCC-D's ability to achieve ideal speed-up and near linear scale-up on large datasets; and (ii) SourcererCC-I: an interactive and real-time version of SourcererCC that is integrated with the Eclipse development environment. SourcererCC-I is built to support developers in clone-aware development and maintenance activities. Finally, this dissertation concludes by presenting two empirical studies conducted using SourcererCC to demonstrate its effectiveness in practice.

Chapter 1

Introduction

Code clone detection locates exact or similar pieces of code, known as clones, within or between software systems. Clones are created for a number of reasons including copy-paste-modify programming practice, accidental similarity in the functionality of the code, plagiarism, and code generation [104]. Code clone detection techniques and tools have long been an area of research as software practitioners depend on them to detect and manage code clones. Clone management is important in order to maintain software quality, detect and prevent new bugs, and also to reduce development risks and costs [104, 103]. Clone research studies also depend on the availability of quality tools [1]. According to Rattan et al., at least 70 diverse tools have been presented in the literature [1]. While several techniques have been proposed for clone detection over many years [89, 53, 94, 60, 6, 11, 74, 54, 69, 76, 58, 73, 93, 26, 122], accuracy and scalability of clone detection tools and techniques still remains an active area of research.

1.1 Motivation

With the amount of source code increasing steadily, large-scale clone detection has become a necessity. Large code bases and repositories of projects have led to several new use cases of clone detection including mining library candidates [51], detecting similar mobile applications [21], license violation detection [73, 39], reverse engineering product lines [44, 39], finding the provenance of a component [31], and code search [65, 64]. While presenting new opportunities for application of clone detection, these modern use cases also pose scalability challenges.

To further illustrate the problem and its scale in practice, consider a real life scenario where a retail banking software system is maintained by Tata Consultancy Services (TCS). A team at TCS deployed the banking system for many different banks (clients) and maintained a separate code base for each of these banks. After following this practice for a while, they decided to form a common code base for all these banks to minimize expenses occurring due to: (i) duplicated efforts to deliver the common features; and (ii) separately maintaining existing common parts of different code bases.

As part of this bigger goal, the team decided to first identify common code blocks across all the code bases. In order to assess the feasibility of using clone detection tools for this task, the team[1] ran CloneDR, an AST based commercial clone detection tool on ML0000, a single COBOL program consisting of 88K LOC. The clone detection process took around 8 hours on an IBM T43 Thinkpad default specification[2] machine. Each bank's code base (8 of them) ran into many million lines of code spanning across thousands of such COBOL programs in different dialects, posing a major scalability challenge. Faced with this challenge, the team made another attempt that was unsuccessful with a tool called Simian before deciding not to pursue this exercise further. More details about the case study are available at [120].

[1]The author was part of the team that carried out the analysis

[2]i5 processor, 8 GB RAM and 500 GB disk storage

This situation at TCS is not unique and in fact it represents the state of many companies in the service industry that are now moving away from the green field development model and adopting the packaging model to build and deliver software. In fact, as Cordy points out, it is a common practice in industry to clone a module and maintain it in parallel [27]. Similarly in open source development, developers often clone modules or fork projects to meet the needs of different clients, and may need large-scale clone detectors to merge these cloned systems towards a product-line style of development.

While the above use cases are more pertinent to industry, researchers are also interested in studying cloning in large software ecosystems (e.g. Debian), or in open-source development communities (e.g. GitHub) to assess its impact on software development and its properties. However, very few tools can scale to the demands of clone detection in very large code bases [115, 103]. For example, Kim and Notkin [67] reflected how they wanted to use clone detection tools for doing origin analysis of software files but were constrained by its speed due to n-to-n file comparison. In his work on using clone detection to identify license violations, Koschke [73] reflects the following: "Detecting license violations of source code requires to compare a suspected system against a very large corpus of source code, for instance, the Debian source distribution. Thus, code clone detection techniques must scale in terms of resources needed". In 2014, Debian had 43,000[3] software packages and approx. 323 million lines of code. In one of their studies to investigate cloning in FreeBSD, Livieri et al. [83] motivate the need of scalable code clone detection tools as follows: "Current clone detection tools are incapable of dealing with a corpus of this size, and might either take literally months to complete a detection run, or might simply crash due to lack of resources."

As a result, the opportunities presented by the modern use cases of clone detection, along with the scalability challenges associated with them, certainly make clone detection an active area of research in the software engineering research community.

[3]https://en.wikipedia.org/wiki/Debian

1.2 Terminology

The dissertation uses the following well-accepted definitions of code clones and their types [13, 104]:

1.2.1 Code Clone Terms

Code Fragment or Block: A continuous segment of source code, specified by the triple (l, s, e), including the source file, l, the start line of the fragment, s, and the end line, e.

Clone Pair: A pair of code fragments that are similar, specified by the triple (f_1, f_2, ϕ), including the similar code fragments f_1 and f_2, and their clone type ϕ.

Clone Class: A set of code fragments that are similar. Specified by the tuple $(f_1, f_2, ..., f_n, \phi)$. Each pair of distinct fragments is a clone pair: $(f_i, f_j, \phi),\ i, j \in 1..n,\ i \neq j$.

Intra-Project Clone: A clone pair where code fragments f_1 and f_2 are found in the same software system.

Inter-Project Clone: A clone pair where code fragments f_1 and f_2 are found in different software systems.

Near-miss Clone: Code fragments that have minor to significant editing differences between them.

1.2.2 Code Clone Types

Type-1 (T1): Identical code fragments, except for differences in white-space, layout, and comments. Figure 1.1 shows an example of a Type-1 clone pair.

```
if (a>=b) {
    c=d+b; // Comment 1
    d=d+1;
} else
    c=d-a; // Comment 2
```

```
if (a >= b)
{
    c = d + b; // MyComment 1
    d = d + 1;
}
else
    c = d - a; // MyComment 2
```

Figure 1.1: Type 1 Example Clone-pair

Type-2 (T2): Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences. Figure 1.2 shows an example of a Type-2 clone pair.

```
if (a>=b) {
    c=d+b; // Comment 1
    d=d+1;
} else
    c=d-a; // Comment 2
```

```
if (a >= y)
{
    x = d + y; // MyComment1
    d = d + 10;
}
else
    x = d      a; // MyComment2
```

Figure 1.2: Type 2 Example Clone-pair

Type-3 (T3): Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences. Figure 1.3 shows an example of a Type-3 clone pair.

```
if (a>=b) {
    c=d+b; // Comment 1
    d=d+1;
} else
    c=d-a; // Comment 2
```

```
if (a >= y)
{
    x = d + y; // MyComment1
}
else {
    x = d - a - 10; // MyComment2
}
```

Figure 1.3: Type 3 Example Clone-pair

Type-4 (T4): Syntactically dissimilar code fragments that implement the same functionality. They are also known as semantic or functional clones. Figure 1.4 shows an example of a Type-4 clone pair.

```
int i, j=1;
for (i=1; i<=VALUE; i++)
    j=j*i;
```

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Figure 1.4: Type 4 Example Clone-pair

This dissertation is limited to the first three types of clones. Type-4 or semantic clones are beyond the scope of this dissertation.

1.3 Problem Statement

While a few novel algorithms [50, 51, 83] in the last decade demonstrated scalability, they do not support Type-3 near-miss clones, where minor to significant editing activities might have taken place in the copy/pasted fragments. These tools therefore miss a large portion of

the clones, since there are more number of Type-3 clones in the repositories than the other types [103, 106, 114]. Furthermore, the ability to detect Type-3 clones is most needed in large-scale clone detection applications [103, 65, 21].

Many techniques have also been proposed to achieve a few specific applications of large-scale clone detection [73, 65, 21], however, they make assumptions regarding the requirements of their target domain to achieve scalability. For example, detecting only file-level clones to identify copyright infringement, or detecting clones only for a given block (clone search) in a large corpus. These domain-specific techniques are not described as general large-scale clone detectors, and face significant scalability challenges for general clone detection.

The scalability of clone detection tools is also constrained by the computational nature of the problem itself. A fundamental way of identifying if two code blocks are clones is to measure the degree of similarity between them, where similarity is measured using a similarity function. A higher similarity value indicates that code blocks are more similar. Thus we can consider pairs of code blocks with high similarity value as clones. In other words, to detect all the clones in a system, each code block has to be compared against every other code block (also known as candidate code blocks), bearing a prohibitively $O(n^2)$ time complexity. Hence, it is an algorithmic challenge to perform this comparison in an efficient and scalable way. This challenge, along with modern use cases and today's large systems make large-scale code clone detection a difficult problem.

1.4 Research Questions

The problem stated above can be characterized in the form of the following research questions.

Research Question 1. [Design] - How can we be more robust to modifications in cloned

code to detect Type-3 clones?

Research Question 2. [**Computational Complexity**] - How can we reduce the $O(n^2)$ candidate comparisons to $O(c.n)$, where $c << n$?

Research Question 3. [**Engineering**] - How can we make faster candidate comparisons without requiring much memory?

Research Question 1 has direct implication on the accuracy of the clone detection technique, and Research Questions 2 & 3 focus on improving the scalability and efficiency of the clone detection technique.

1.5 Thesis

The above research questions motivate the need for clone detection techniques and tools that satisfy the following requirements: (1) accurate and fast detection of near-miss clones, where minor to significant editing changes occur in the copy/pasted fragments; (2) scalability to hundreds of millions of lines of code and several thousand projects without requiring special hardware; and (3) minimal or no dependency on programming languages.

To that effect, I propose SourcererCC, an accurate, near-miss clone detection tool that scales to hundreds of millions of lines of code (MLOC) on a single standard machine. The core idea of SourcererCC is to build an optimized index of code blocks and compare them using a simple bag-of-tokens[4] representation, which is resilient to Type-3 changes. SourcererCC uses several filtering heuristics from the Information Retrieval domain to reduce the size of the index, which in turn, significantly reduces the number of code block comparisons to detect clones. SourcererCC also exploits the ordering of tokens in a code block to measure a live upper-bound on the similarity of code blocks in order to reject or accept a clone candidate

[4]Similar to the popular bag-of-words model [128] in Information Retrieval

with minimal token comparisons.

I can now state my thesis as follows.

SourcererCC improves the state-of-the-art in code clone detection by being the most scalable technique known so far, with accuracy on a par with the current state-of-the-art tools.

CCFinderX [61], Deckard [55], iClones [40], and NiCad [28] are described as the state-of-the-art in the literature as well as in recent clone detection benchmarking experiments [117, 118]. CCFinderX is the most popular and successful tool that has been used in many clone studies. Deckard, iClones and NiCad are popular examples of modern clone detection tools that support Type-3 clone detection. I evaluate the scalability, execution time and accuracy (precision and recall) of SourcererCC against these four state-of-the-art tools.

To measure scalability, I evaluate the performance of the tools on a large inter-project software repository IJaDataset-2.0 [3] consisting of 25,000 projects containing 3 million files and 250 MLOC.

To measure recall, I use two recent benchmarks: (1) a benchmark of real clones, BigCloneBench, that spans the four primary clone types and the full spectrum of syntactical similarity in three different languages (Java, C, and C#); and (2) a Mutation/Injection-based framework of thousands of fine-grained artificial clones. In order to measure precision, I conduct blind user studies where five software engineering researchers validate 2,000 clone pairs reported by the clone detection tools.

1.6 Contributions

In the process of investigating the above claims, this dissertation makes the following contributions:

(i) An accurate and fast approach to clone detection that is both scalable to very large software repositories and robust against code modifications. (Chapter 3)

(ii) SourcererCC, an accurate and fast clone detection tool based on the above approach that scales to hundreds of millions of lines of code on a single standard machine. (Chapter 3)

(iii) Experiments to evaluate performance (scalability and execution time) and quality (recall and precision) of SourcererCC against existing state-of-the-art tools. The results of these experiments can be used as benchmarks for future clone detection tools and techniques. (Chapter 4)

(iv) SourcererCC-D, a distributed version of SourcererCC that exploits the inherent parallelism in the SourcererCC's approach to efficiently scale to large software repositories on a cluster of machines. (Chapter 5)

(v) SourcererCC-I, an interactive version of SourcererCC that is integrated with Eclipse IDE to help developers in clone-aware development and maintenance activities. (Chapter 6)

(vi) Large-scale empirical studies conducted using SourcererCC to understand the relationship between code quality (software metrics and bug patterns) and code clones. This demonstrates SourcererCC's applicability in practice. (Chapter 7)

Chapter 2

Clone Detection: Background and Related Work

Code cloning has been a topic of inquiry for a long time and there is a large amount of work done in this area. In this chapter, we review the research work in clone detection by summarizing some of the most comprehensive surveys conducted in this field [104, 107, 71]. First, we discuss various reasons for the existence of clones. Second, we discuss the implications of code cloning in software systems. Third, we highlight various applications of clone detection research in the context of software engineering. Next, we review various clone detection techniques and tools and discuss their strengths and limitations. Finally, we conclude by reviewing the studies that assess the impact of code cloning on software systems.

2.1 Why Do Code Clones Exist?

While clones can be accidental, most often several factors contribute to the creation of code clones. Below we summarize the main factors that lead to cloning in software systems,

according to the literature.

1. Cloning as a Way to Reuse

One of the most prominent ways to reuse code is through the copy-paste-modify programming practice. This practice, in turn, creates clones. Kasper and Godfrey [63] found clones in systems with similar functionality that later diverged significantly as the systems evolved. They noticed that the developers often fork the repository of existing similar solutions and then add or modify the code to adapt to the task at hand (e.g. in case of device drivers). Furthermore, developers often face issues in understanding software systems in a new domain. This forces them to use the example-oriented programming by copying and adapting code already written in the system.

2. Cloning for Maintenance Benefits

Clones are introduced in the systems to reduce the risk of new code breaking the system. Cordy [27] describes that in some domains (e.g. Banking and Insurance) developers are often asked to reuse the existing code by copying and adapting it to the new product requirements because existing code is already well tested.

In many real time applications and also in the financial domain (mainframe programs), monolithic code is preferred over modularized code, as function calls are expensive. This practice often leads to code duplication for performance reasons.

Sometimes clones can also be desirable due to separation of concerns. Rieger [101] points out that since two cloned code fragments are independent of each other, they can evolve at different paces providing separation of concerns. This might be desirable in the absence of effective test strategies.

As software evolves, even code written with good abstraction may no longer represent a single, common abstraction, but instead become condition-laden procedure which interleaves a number of functionalities. In such scenarios, developers re-introduce duplication by in-

lining the abstracted code back into every caller to make the code maintainable.

3. Limitation of Programming Languages/Frameworks

Clones can be introduced due to the limitations of programming languages, especially when the languages in question do not have sufficient abstraction mechanisms. For example, many procedural language do not have features like inheritance, generic types, templates, and parameter passing, making it difficult to write reusable code [10].

Clones can also be introduced by frameworks that use code generators to automatically generate starter code. Some examples include generating getters and setters methods for class attributes in Java, or generating classes to access database fields.

4. Software Development Practices

Software development practices often influence how code is written. For example, in practice, developers are often subjected to pressure due to release deadlines in a project. These deadlines often force developers to solve problems at hand by copy-pasting existing code and adapting to current needs. Moreover, often in certain institutions or companies, developers do not have 'write' access to the reused code for policy reasons. In such cases, copying code, and modifying it, is the only available option [27].

5. Cloning by Chance (Accidental Cloning)

Clones may also be introduced unintentionally. For example, libraries or APIs often have a specific ordering of function calls or tasks to be performed to use them [63]. Such library usages may introduce accidental clones. Also, coincidentally, different developers may implement the same logic in a similar way creating accidental clones.

2.2 Issues Due to Code Cloning

Code cloning can have severe impacts on the quality, re-usability and maintainability of a software system. Below we list some of the well-known issues of having clones in the system.

(i) Code cloning may increase the probability of bug propagation if the original code fragments contain a bug [81].

(ii) The process of cloning a code fragment can be error prone and may introduce new bugs in the system [11]. For example, developers may accidentally miss updating all the references of the modification often leading to inconsistent and incomplete changes in the copied fragment.

(iii) If a bug is found in the cloned code, all of its similar counterparts need to be checked if they contain a bug as it is likely that the bug may already be present at the time of reuse [92].

(iv) Cloning may also increase the size of a software. This may be an issue for systems with hardware constraints, as hardware might need an upgrade with the increase in size of the software [63].

(v) Cloning may break design abstractions or indicate lack of inheritance. Thus, parts of code with cloning are sometimes difficult to reuse [92].

2.3 Applications of Clone Detection

Apart from maintenance benefits, there are several other benefits and applications of detecting clones listed as follows:

1. Plagiarism Detection. Plagiarism detection is the process of locating instances of plagiarism within source code. It is one of the areas where many clone detection techniques have been effectively used [86, 52, 23].

2. Library or API detection. Burd and Munro [17] and Ishihara et al. [51] reflect that frequent cloning of files or large code blocks may help in identifying potential candidates for forming a libarary or an API.

3. Software Provenance Analysis. Clone detection can also be useful for identifying the origin of software components. Julius et al. [31] recover provenance of software entities for technical and ethical concerns using code clone detection in large software repositories.

4. Multi-version Program Analysis. Kim and Notkin [67] used clone detection as one of the techniques to match program element across multiple versions to carry out Multi-version program analyses.

5. Program Understanding. Clone detection can also be useful for program comprehension. For example, if the functionality of a code block is known, it helps to comprehend the functionalities of other classes containing similar copies of that block [101].

6. License Violation and Copyright Infringement. Software copyright is used by proprietary companies or even many open source foundations (e.g. Apache) to prevent the unauthorized copying of their software. It is one of the most popular and practical uses of modern clone detectors. Koschke [73] used a suffix tree based clone detection technique to identify license violations by querying a code block against a large corpus of open source software systems having different licenses.

7. Reverse Engineering Product Line Architecture. This is another modern use case of clone detectors. The goal is to identify commonalities in the code bases to reverse engineer the common features in the product and its derivatives. Hemel and Kosckhe [44] used code

clone detection techniques to identify the issues in the Linux code base because of increasing fragmentation of Linux derivatives.

8. Code Search. Recently researchers have also used cloning for assisting code search [65, 64]. Keivanloo et al. [66] engage real-time clone detection within their code search process to improve their performance. They use code blocks from the initial few results of a code search query to detect similar code blocks in order to improve the recall of code search.

Detection of license violation, reverse engineering of product lines, and code search are some of the modern use cases that require clone detectors to be highly scalable to very large datasets.

2.4 Clone Detection Techniques and Tools

Several techniques have been proposed for clone detection over many years [89, 53, 94, 60, 6, 11, 74, 54, 69, 76, 58, 73, 93, 26, 122]. These techniques differ in many dimensions ranging from the type of detection algorithm they use to the source code representation they operate on. Techniques using various representations include Tokens [60, 6], Abstract Syntax Trees (AST) [11, 74, 54], Program Dependence Graphs [69, 76], Suffix Trees [11, 58, 60, 73], Text representations [28, 87, 26], and Hash representations [122]. Each of these different approaches have their own merits and are useful for different use cases. For example, AST based techniques have high precision, and are useful for refactoring of clones, but may not scale. Moreover, token based techniques have high recall but may yield clones which are not syntactically complete [14]. They are useful where high recall is important. An excellent survey highlighting the strength and limitation of various clone detection techniques is available at [107].

Below we provide a brief description of each of these techniques.

1. Text or String-based Techniques

Text/String-based techniques use basic text transformation (e.g. stop words removal, stemming, etc.) on source code and use string matching algorithms to detect clones. These techniques mostly differ from one another at: (i) the granularity of the matching unit; and (ii) the string matching algorithms. For example, Cordy et al.'s NiCad [28] finds near-miss clones using an efficient text line comparison technique based on longest common subsequence algorithm. Similarly, Marcus and Maletic [87] apply Latent Semantic Indexing (LSI) on function text to identify high level concept clones.

2. Token-based Techniques

Token-based techniques differ from string-based techniques in that they use more sophisticated transformation on the source code to construct a token stream. They mostly use a lexical analyser to parse the source code, apply rule-based transformation, and generate a stream of tokens. The presence of such tokens makes it possible to detect code portions that have different syntax but have similar meaning and also filter out code portions with specified structure patterns.

Baker's *Dup* [6] and Kamiyo et al.'s CCFinderX [61] are two popular token-based clone detectors. SourcererCC also uses this approach although with a very light-weight lexer to have minimal dependency on the programming language.

3. Abstract Syntax Tree-based Techniques

AST-based techniques transform the source code into a parse tree and use graph or tree matching algorithms to detect similar subtrees [11, 55, 74]. While these techniques are precise, they often face scalability issues as parse trees are rich in information and hence consume a high amount of memory. AST-based clone detectors are a popular choice for building clone removal/refactoring tools.

4. PDG-based Techniques

PDG-based approaches work in a way similar to AST-based techniques except that they represent the program in the form of a program dependence graph. PDG contains the control flow and data flow information of a program and hence it carries more semantic information than an AST. As a result, PDG-based approaches are more robust to insertion and deletion of code, non-contiguous code, reordered statements, but similar to AST-based techniques they are not scalable to large programs.

Komondoor and Horowitz [70] were the first to use PDG to detect clones. They represent a program as a dependency graph, and transform the problem of clone detection into finding isomorphic subgraphs over PDG. Krinke [77] extended their work to show how the k-length pattern matching algorithm can be used for detecting maximal similar subgraphs in a more efficient way.

5. Metrics-based Techniques

Metrics-based techniques characterize code fragments using a set of metrics. Code fragments with similar metric values are identified as code clones. Cyclomatic complexity, function points, lines of code, etc. are examples of possible metrics. Mayrand et al. [90] calculate 21 metrics for each function unit of a program and identify functions with similar metric values as code clones. They found that metrics-based technique is more effective for detecting clones at a high level of granularity (e.g. class or file-level).

6. Hybrid Approaches

Researchers have also combined multiple techniques or program representations to achieve hybrid techniques for clone detection. For example, Jiang et al. [55] propose a hybrid approach to detect clones using AST and Local Sensitive Hashing technique (LSH). They first compute vectors representing the structural information within ASTs in the Euclidean space and then use LSH to cluster similar vectors. Vectors in the same cluster are considered clones. They found that using LSH to compare vectors is much more efficient than compar-

ing the AST nodes. Koschke et al. [75] make use of token-based suffix trees to represent the information in the ASTs. Hence, instead of comparing AST nodes directly, they compare nodes of suffix tree representing AST nodes. This hybridization scales linearly in time, which makes it very attractive for large systems.

Many of the above code clone techniques have been useful in software maintenance use cases where a fragment of the code or smaller subset of the code clones have to be detected in a project [107, 96, 53]. While previous research has mainly focused on identifying code fragments on a per-project basis, several new use cases have emerged, identifying the need for detecting all the clones in the system or a family of systems. These use cases including detection of license violation [73, 39], reverse engineering the product lines [44, 39], finding the provenance of a component [31], and even code search [65, 64] have redefined scalability. This presents new research challenges for code clone detection techniques - How to scale to large repositories while also being accurate?

2.5 Measures to Evaluate Clone Detection Techniques

There are several measures with which the tools can be evaluated or compared against each other. These measures not only help evaluate the tools but also allow users to pick the right one for a particular purpose of interest. These measures are also desirable properties of clone detection tools and techniques.

Below we list some of the well-known measures used in the literature for comparing different clone detection tools or techniques [55, 107, 109, 53]:

Precision. The tool should not detect instances of code blocks which are not clones i.e., low false positive rate.

Recall. The tool should be able to detect most or all of the clones of the subject system.

Robustness. The tool should be able to detect clones of various clone types including near-miss clones with high precision and recall.

Scalability. The tool should be able to detect clones in large software systems and repositories with reasonable memory usage.

Execution Time. The tool should detect clones in a reasonable amount of time depending on the size of the input.

Portability. The tool should have minimal dependency on the target platform or language and should be easy enough to adapt to various languages and dialects.

2.6 Impact of Code Cloning on Software Systems

Impact of code cloning on software systems has long been a topic of inquiry. Traditionally cloning has always been looked upon as a bad practice in the context of software maintenance. While there are many negative effects of code cloning (see Section 2.2) [57, 38, 84, 77, 127, 59], the widespread presence of clones has motivated researchers to dig deeper and understand the usage scenarios.

Kasper et al. [63] presented eleven patterns by examining clones in two systems. They found out that not all usage patterns have negative consequence and some may even have positive consequence on quality.

Ossher et al. [96] looked at circumstances of file cloning in open source Java systems and classified the cloning scenarios into good, bad, and ugly. These scenarios included good use cases like extension of Java classes and popular third-party libraries, both large and small. They also found ugly cases where a number of projects occur in multiple online repositories, or have been forked, or were copied and divided into multiple subprojects. From a software engineering standpoint, some of these situations are more legitimate than others.

Kim et al. [68] studied clone evolution in two open source systems. They found that most of the clone pairs are short lived and about 72% of the clone pairs diverge within eight commits in the code repository. They found that several clones exist by design and cannot be refactored because of the limitation of programming language or it would require a design change. To that end, de Wit et al. [32] proposed CLONEBOARD, an Eclipse plug-in implementation to track live changes in clones and offering several resolution strategies for inconsistently modified clones. They conducted a user study and found that developers see the added value of the tool but have strict requirements with respect to its usability.

Cordy [25] analyzes clones and intentions behind cloning of a financial institution system and argues that external business factors may facilitate cloning. He notes that financial institutions avoid situations that can break the existing code under any circumstances. Abstractions might introduce dependencies, and modifying such abstractions induces the risk of breaking existing code. Cloning minimizes this risk as code is maintained and modified separately, localizing the risk of errors to a single module. Similarly, Rajapakse et al. [100] found that reducing duplication in a web application only had negative effects on the modifiability of an application. He notes that after significantly reducing the size of the source code, a single change required testing of a vastly larger portion of the system.

Rahman et al. [98] investigate the effect of cloning on defect proneness on four open source systems. They looked at the buggy changes and explored their relationship with cloning. They did not find evidence that cloned code is riskier than non-cloned code.

Brutnik et al. [16] use clone detection techniques in a novel way to find cross-cutting concerns in the code. They manually identify five specific cross-cutting concerns in an industrial C system and analyze to what extent clone detection is capable of finding them. The initial results favorable and imply that clone detectors can certainly be used for building automated "aspect miner".

Sajnani et al. [110] conducted a comparative analysis of bug patterns in Java cloned and non-cloned code on 31 Apache Java projects. They found that the defect density of cloned code is in fact less 2 times less than of non-cloned code. Moreover, they found that about 75% of the times, a bug pattern is also duplicated when the code is cloned.

2.7 Chapter Summary

This chapter provided a brief overview of the research in the area of code clone detection. Clone detection is an active area of research and there is a large amount of work in the literature spanning from creating taxonomy of clones, to understanding the reasons behind it, to developing tools and techniques to detect, remove and even manage clones. There are measures designed and borrowed from information retrieval field to compare and evaluate tools against each other.

While traditionally clone detection tools have been primarily designed for maintenance tasks, several modern use cases have posed new challenges in this area. This has revived the interest of the community in building accurate and scalable clone detection tools and techniques that can efficiently scale to large software repositories and systems. It is also the motivation behind the development of SourcererCC and other tool suites presented in this dissertation.

Chapter 3

SourcererCC: Accurate and Scalable Code Clone Detection

Part of the material in this chapter is included with the permission of the IEEE and based on our work in:

- Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.; Lopes, C., "SourcererCC: Scaling Code Clone Detection to Big-Code," International Conference on Software Engineering (ICSE'16), May 2016

This chapter introduces SourcererCC, a token-based accurate near-miss clone detector that exploits an optimized index to scale to hundreds of millions of lines of code (MLOC) on a single machine. SourcererCC compares code blocks using a simple and fast bag-of-tokens strategy which is resilient to Type-3 changes. We describe SourcererCC's clone detection process in detail below.

3.1 Problem Formulation

This section describes how SourcererCC formulates the problem of code clone detection.

A software project P is represented as a set of code blocks $P : \{B_1, ..., B_n\}$. As defined earlier, a code block is a continuous segment of source code (e.g. method or functions, classes, code between). A code block B, in turn, is represented as a bag-of-tokens (multiset) $B : \{T_1..., T_k\}$. A token is represented by a programming language keyword, literal, or an identifier. There is no preprocessing of tokens except, a string literal is further split on whitespace and the operators (e.g. $'*'$, $'+'$) are discarded. Since a code block may have token multiplicity, each token is represented as a $(token, frequency)$ pair where $frequency$ denotes the number of times a $token$ appears in a given code block. This step further reduces a code block representation to a set of $(token, frequency)$ pairs.

In order to quantitatively infer if two code blocks are clones, we use a similarity function which measures the degree of similarity between code blocks, and returns a non-negative value. The higher the value, the greater the similarity between the code blocks. As a result, code blocks with similarity value higher than the specified threshold are identified as clones.

Formally, given two projects P_x and P_y, a similarity function f, and a threshold θ, the aim is to find all the code block pairs (or groups) $P_x.B$ and $P_y.B$ $s.t$ $f(P_x.B, P_y.B) \geq \lceil\, \theta \cdot \max(|P_x.B|, |P_y.B|)\, \rceil$. Note that for intra-project similarity, P_x and P_y are the same. Similarly, all the clones in a project repository can be revealed by doing a self-join on the entire repository itself.

Similarity Function: We compute similarity by measuring overlap between the code blocks. For example, given two code blocks B_x and B_y, the overlap[3] similarity $OS(B_x, B_y)$

[3]SourcererCC can be adapted for Jaccard and Cosine similarity functions as well.

is computed as the number of tokens shared by B_x and B_y.

$$OS(B_x, B_y) = |B_x \cap B_y| \tag{3.1}$$

In other words, if θ is specified as 0.8, and $max(|B_x|, |B_y|)$ is t, then B_x and B_y should share at least $\lceil 0.8|\dot{t}| \rceil$ tokens to be identified as a clone pair. Note that if a token a appears in B_x twice and thrice in B_y, the match between B_x and B_y due to token a is two i.e., the minimum of its occurrence in either of the code blocks.

Example: To illustrate, Figure 3.2 shows two methods from *DOMTransformer.java* in *org.apache.cocoon.transformation* package of Apache Cocoon project. For the ease of representation, Table 3.1 shows the mapping of source terms with shorter terms. Using this mapping, the methods can be transformed into the following sets of (token, frequency) pair (B_1 & B_2):

$$\begin{aligned} B_1 = \{&(T1,1),(T2,1),(T3,2),(T4,1),(T5,2),(T6,2),\\ &(T7,2),(T8,2),(T9,1),(T10,1),(T11,5),(T12,2),\\ &(T13,2),(T14,1),T(15,1),(T16,1)\} \end{aligned}$$

$$\begin{aligned} B_2 = \{&(T1,1),(T2,1),(T17,2),(T4,1),(T5,2),(T6,2),\\ &(T7,2),(T8,2),(T9,1),(T10,1),(T11,5),(T12,2),\\ &(T13,2),(T14,1),T(15,1),(T16,1)\} \end{aligned}$$

Figure 3.1: Code blocks represented as a set of (token, frequency) pairs

The size of each method is $t = 27$ tokens, out of which they share 25 tokens. Given the overlap similarity function, and θ specified as 0.8, the minimum number of tokens they should share in order to be identified as clones is 22 ($\lceil \theta|\dot{t}| \rceil = 22$). Since they share 25 terms (> 22), they are clones for a given θ. However, if we change θ to 0.95, the minimum number of tokens shared should be 26. Hence, in that case, the two methods will not be identified as clones. Therefore, the higher the value of θ, the stricter the similarity between the methods.

```
1. public void characters(char c[], int start, int len)
2.   throws SAXException {
3.   this.stack.setOffset(this.oldOffset);
4.   this.consumer.characters(c, start, len);
5.   this.stack.setOffset(this.newOffset);
6. }
```

```
1. public void ignorableWhitespace(char c[], int start, int len)
2.   throws SAXException {
3.   this.stack.setOffset(this.oldOffset);
4.   this.consumer.ignorableWhitespace(c, start, len);
5.   this.stack.setOffset(this.newOffset);
6. }
```

Figure 3.2: Methods from Apache Cocoon Project

Source Term	Mapping
public	T1
void	T2
characters	T3
char	T4
c	T5
int	T6
start	T7
len	T8
throws	T9
SAXException	T10
this	T11
stack	T12
setOffset	T13
oldOffset	T14
consumer	T15
newOffset	T16
ignorableWhiteSpace	T17

Table 3.1: Source Term Mapping

In order to detect all clone pairs in a project or a repository, the above approach of computing the similarity between two code blocks can simply be extended to iterate over all the code blocks and compute pairwise similarity for each code block pair. For a given code block, all the other code blocks compared are called candidate code blocks or candidates in short.

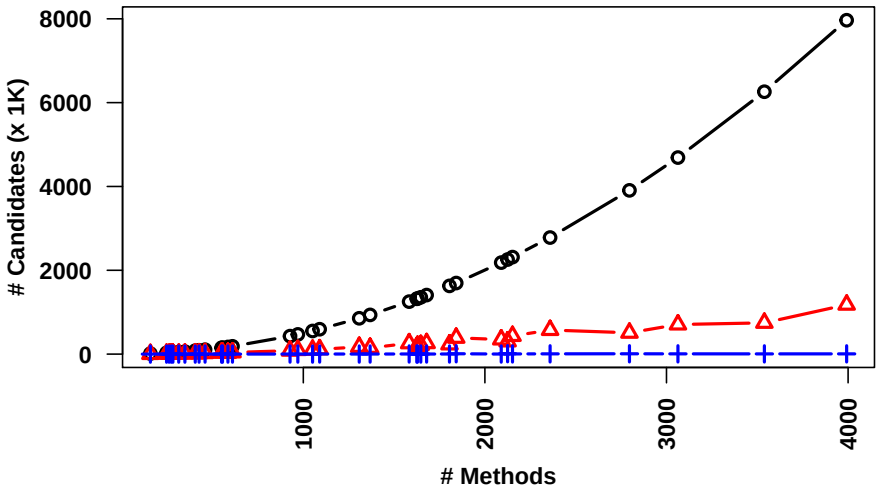

Figure 3.3: Growth in number of candidate comparisons with the increase in the number of code blocks

While the approach is very simple and intuitive, it is also subject to a fundamental problem that prohibits scalability - $O(n^2)$ time complexity. Figure 3.3 describes this by plotting the number of total code blocks (X-axis) vs. the number of candidate comparisons (Y-axis) in 35 Apache Java projects (see Appendix A for details about the projects). Note that the granularity of a code block is taken as a method. Points denoted by $\circ$ show how the number of candidates compared increases quadratically[1] with the increase in number of methods. SourcererCC uses advanced index structures and filtering heuristics as described in the next few sections to significantly reduce the number of candidate comparisons during clone detection.

[1]The curve can also be represented using $y = x(x-1)/2$ quadratic function where x is the number of methods in a project and y is the number of candidate comparisons carried out to detect all clone pairs.

3.2 Overview of the Approach

SourcererCC's general procedure is summarized in Figure 3.4. It primarily operates in three stages: (i) code block creation; (ii) partial index creation; and (ii) clone detection.

SourcererCC's first step is to parse the source code, extract code blocks according to the granularity at which the clones are to be detected. The granularity of extraction could be file-level, method-level or block-level. The extracted code blocks are then tokenized.

To accomplish this, SourcererCC uses a simple scanner that is aware of token and block semantics of a given language[2]. The extracted code blocks are represented as a set of $(token, frequency)$ pairs as already shown in Figure 3.1. This process is described as Code Block Creation (step 1) in Figure 3.4.

In the next step, namely, Partial Index Creation (step 2 in Figure 3.4), SourcererCC builds an inverted index mapping tokens to the code blocks that contain them. An inverted index (also referred to as postings file or inverted file) is an index data structure storing a mapping from content, such as words or numbers (in our case tokens), to its locations in a document or a set of documents (in our case code block). Unlike previous approaches [50], SourcererCC does not create an index of all the tokens in a code blocks, instead it uses a filtering heuristic (Section 3.3.1) to construct a partial index of only a subset of the tokens in each block.

In the Clone Detection phase (step 3 in Figure 3.4), SourcererCC iterates through all of the code blocks, and retrieves their candidate clone blocks from the index. As per the filtering heuristic, only a subset of the tokens in a code block are used to query the index, which reduces the number of candidate blocks. After candidates are retrieved, SourcererCC uses another filtering heuristic (Section 3.3.2), that exploits ordering of the tokens in a code block to measure a live upper-bound and lower-bound of similarity scores between the query and

[2]Currently SourcererCC supports Java, C and C# using TXL [24], but it can be easily extended to other languages.

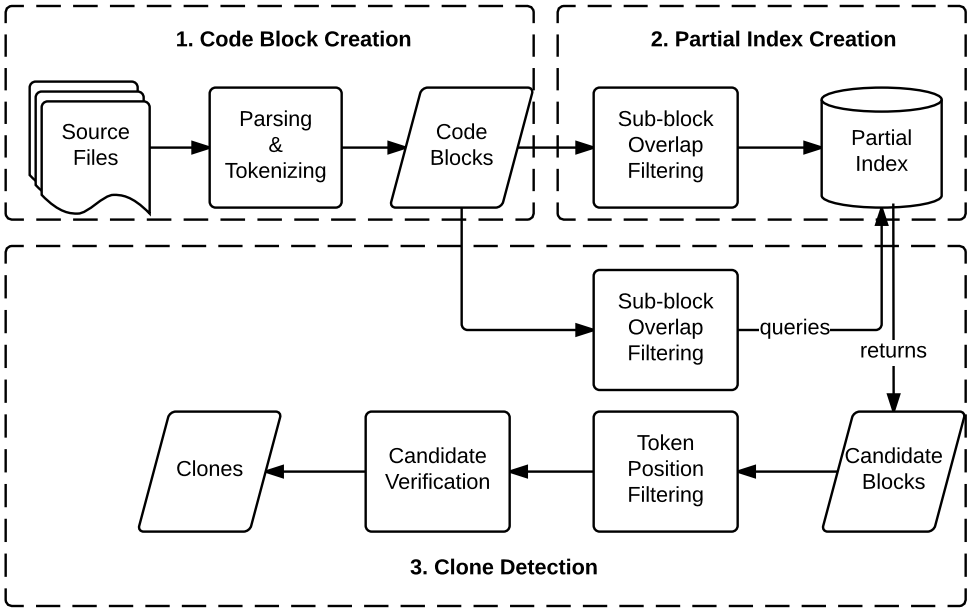

Figure 3.4: SourcererCC's clone detection process

candidate blocks. Candidates whose upper-bounds fall below the similarity threshold are eliminated immediately without further processing. Similarly, candidates are identified as clones as soon as their lower-bounds exceed the similarity threshold. This is repeated until all the clones of every code block are located. SourcererCC also exploits symmetry to avoid detecting the same clone pair twice.

In the following sections, we provide a detailed description of the filtering heuristics and overall clone detection algorithm.

3.3 Filtering Heuristics to Reduce Candidate Comparisons

This section describes filtering heuristics that enable SourcererCC to effectively reduce the number of candidate comparisons during clone detection. These heuristics are inspired by the work of Chaudhuri et al. [20] and Xiao et al. [126] on efficient set similarity joins in databases.

3.3.1 Sub-block Overlap Filtering

Sub-block overlap filtering follows an intuition that when two sets have a large overlap, even their smaller subsets should overlap. Since we represent code blocks as bag-of-tokens (i.e., a multiset), we can extend this idea to code blocks, i.e., when two code blocks have large overlap, even their smaller sub-blocks should overlap. Formally, we can state it in the form of a following property:

Property 1.a: *If blocks B_x and B_y consist of t tokens each, and if $|B_x \cap B_y| \geq i$, then any sub-block of B_x consisting of $t - i + 1$ terms will overlap with block B_y.*

To illustrate, consider the following two blocks:

$$B_x = \{a, b, c, d, e\}$$

$$B_y = \{f, b, e, d, c\}$$

B_x and B_y have a total of 5 tokens ($t = 5$) each, and an overlap of 4 tokens ($i = 4$). Let S_x

represent a set of all the sub-blocks of B_x of size 2 $(t - i + 1)$:

$$S_x = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, d\}, \{c, e\}, \{d, e\}\}$$

Then, according to Property 1.a, all the elements of S_x will have a non-zero overlap (at least share one token) with B_y.

To understand the implications of this property, consider the above two blocks with 5 tokens each, and let θ be specified as 0.8. That is, the two blocks should share at least $\lceil 0.8 * 5 \rceil = 4$ tokens to be considered as clones of each other. In this context, if we wish to find out if B_x and B_y are clones, then, using Property 1.a, we can infer that all the sub-blocks of B_x should share at least one token with B_y. That is to say, if any sub-block of B_x did not share a token with B_y, we can certainly deduce that B_x and B_y cannot be clones for a given value of θ.

Interestingly, the above principle suggests that instead of comparing all the tokens of B_x and B_y against each other, we could simply compare only the tokens of any sub-block of B_x to identify if B_x and B_y will *not* happen to be clones. In this way, by filtering out a large subset of B_x, we can often reduce the total number of comparisons between B_x and B_y by very significant margins.

The above example applies filtering to only block B_x. Naturally, the question is, if and how can we apply filtering heuristic to both the blocks? It turns out that the filtering property is applicable to both the blocks if the blocks are in the *same* order. Formally, this refinement can be stated in the form of the following property:

Property 1.b: *Given blocks B_x and B_y consisting of t tokens each in some predefined order, if $|B_x \cap B_y| \geq i$, then the sub-blocks SB_x and SB_y of B_x and B_y respectively, consisting of*

first $t - i + 1$ tokens, must match at least one token.

To further understand the implication of this refinement, let us consider the code blocks in the previous example to be alphabetically ordered.

$$B_x = \{\mathbf{a,\ b}, c, d, e\}$$

$$B_y = \{\mathbf{b,\ c}, d, e, f\}$$

.

According to Property 1.b, since the two blocks are now in the same order (alphabetically ordered), in order to find out if B_x and B_y are clones, we can simply check if their sub-blocks consisting of only first $t-i+1 = 2$ tokens match at least one token. In this case, they do, as token b is common in both the sub-blocks (marked in bold). However, if they did not share any token, then even without comparing the remaining tokens in the blocks, we could have most certainly figured that B_x and B_y will not end up as a clone pair for the given θ. In other words, Property 1.b suggests that instead of comparing all the tokens of B_x and B_y against each other, we could simply compare their sub-blocks consisting of only first $t-i+1$ tokens to deduce if B_x and B_y will *not* be clones.

3.3.1.1 Token Ordering in the Code Blocks

As stated in Property 1.b, the sub-block overlap filtering heuristic is applicable only if the tokens in the code blocks follow a predefined global order. While there are many ways in which tokens in a block can be ordered (e.g., alphabetical order, length of tokens, occurrence frequency of token in a corpus), a natural question is what order is most effective in this context. The goal is to pick an ordering that further minimizes the number of comparisons among the code blocks.

As it turns out, software vocabulary follows Zipf's law [130] and exhibit characteristics very similar to natural languages [47]. That is, there are only few very popular (frequent) tokens in the corpus, and the frequency of tokens decreases very rapidly with their rank. In other words, while most of the code blocks are likely to contain one or more of few very popular tokens (e.g. keywords, or common identifier names like i, j, *count*), not many will share rare tokens (e.g. identifiers that are domain or project specific). The basic idea is to eliminate the comparison of frequent tokens and instead, check if two code blocks share rare tokens. Therefore, if the tokens in the code blocks are ordered in the reverse order of their popularity in the corpus, naturally, their sub-blocks will consist of rare tokens because popular tokens will be pushed at the end of the code blocks. Such arrangement will ensure low probability of different sub-blocks sharing similar tokens. As a result, this ordering will eliminate more false positive candidates[3].

To implement this, we create a global token order list consisting of all the tokens in the corpus[4]. First, a list of all the tokens in the corpus is computed. Next, the corpus frequency [5] of each token in the list is calculated. The list is sorted in the ascending order of the corpus frequency of tokens. Tokens with lower frequency appear at the top of the list whereas, the tokens with higher frequency appear at the bottom of the list. The resultant global token order list is used to order the tokens in the candidate code block. Since high frequency tokens will have lower ranking in the global token order list, they will appear at the end in the code blocks. Therefore, the sub-blocks consisting of first few tokens will mostly consist of rare tokens - subsequently decreasing the probability of a token match with candidate sub-blocks.

[3]Candidate code blocks that will not result into a clone pair are known as false positive candidates.
[4]Corpus consist of a project or a repository on which clone detection is to be performed
[5]Corpus frequency of a token is a measure of how many times a token appeared in the entire corpus

3.3.1.2 Computing the Correct Sub-block Length

The sub-blocks consist of only the first few tokens of the block. The natural question is how many tokens should be considered or in other words, what should be the length of sub-blocks? It is important to note that the length of a sub-block has direct implication on filtering the candidates. Hence, if computed incorrectly, it can eliminate blocks which could actually result into clones, i.e., lowering the recall of the technique. We can compute the correct sub-block length in the following way:

Using Property 1.b, we know that the minimum number of overlapping tokens between the two code blocks is:

$$t - i + 1 \tag{3.2}$$

Now, given the similarity threshold value θ and the number of tokens in the code block as t, the minimum number of overlapping tokens can be computed as:

$$\lceil \theta | \dot{t} | \rceil \tag{3.3}$$

Comparing Equation 3.2 and Equation 3.3, the length of sub-block, i, is computed as:

$$i = |t| - \lceil \theta | \dot{t} | \rceil + 1 \tag{3.4}$$

Hence, in a code block of t tokens, and a similarity threshold value of θ, a sub-block length of $|t| - \lceil \theta | \dot{t} | \rceil + 1$ will ensure that no potential clones will be missed.

Note that the length of a sub-block depends on the size of the block for which it is computed and hence cannot be the same for all the blocks. Moreover, in most cases, the two blocks compared will have different sizes, say t_1 and t_2. However, in such cases, the formula to

compute the length does not change, except $|t|$ is now computed as $|t| = max(|t_1|, |t_2|)$.

Revisiting the Figure 3.3, the points denoted by △ show the number of candidate comparisons after applying the sub-block overlap filtering. The large difference from the earlier curve (∘) shows the impact of filtering in eliminating candidate comparisons.

The below section discusses when the use of Property 1 may still be ineffective and demonstrates how ordering of tokens in a code block can be further exploited to formalize yet another filtering heuristic that is extremely effective in eliminating even more candidate comparisons.

3.3.2 Token Position Filtering

In order to understand when Property 1 may be ineffective, consider code blocks B_x and B_y from the previous example, except B_x now has one fewer token. Hence $B_x = \{\mathbf{a,\ b}, c, d\}$ and $B_y = \{\mathbf{b,\ c}, d, e, f\}$.

Assuming the same value of θ, the blocks must still match 4 tokens ($\lceil\, \theta \cdot \max(|B_x|, |B_y|)\, \rceil = \lceil 0.8{*}5 \rceil = 4$) to be a clone pair. But since the two blocks have only 3 tokens in common, they cannot be identified as a clone pair. However, note that their sub-blocks (shown in bold) consisting of first $t - i + 1 = 2$ tokens still have a common token b. As a result, *Property 1* is satisfied and B_y will be identified as a candidate of B_x, although B_x and B_y eventually will not end up as a clone pair. In general, cases when the code blocks have fairly different sizes it is likely that they may result into false positives even after satisfying Property 1.

Interestingly, to overcome this limitation, the ordering of tokens in code blocks can be exploited. For example, if we closely examine the position of the matched token b in B_x and B_y, we can obtain an estimate of the maximum possible overlap between B_x and B_y as the sum of *current matched tokens* and the *minimum number of unseen tokens in* B_x *and*

B_y, i.e., $1 + min(2, 4) = 3$. Since this upper-bound on overlap is already smaller than the needed threshold of 4 tokens, we can safely reject B_y as a candidate of B_x. Note that we can compute a safe upper-bound (without violating the correctness) because the blocks follow a predefined order. The above heuristic can be formally stated as follows.

Property 2: *Let blocks B_x and B_y be ordered and $\exists$ token t at index i in B_x, s.t B_x is divided in to two parts, where $B_x(first) = B_x[1...(i-1)]$ and $B_x(second) = B_x[i...|B_x|)]$ Now if $|B_x \cap B_y| \geq \lceil \theta \cdot \max(|B_x|, |B_y|) \rceil$, then $\forall\ t \in B_x \cap B_y$, $|B_x(first) \cap B_y(first)| + min(|B_x(second)|, |B_y(second)|) \geq \lceil \theta \cdot \max(|B_x|, |B_y|) \rceil$*

Revisiting the Figure 3.3, the points denoted by + show the number of candidate comparisons after applying the token position filtering. The reduction is so significant that empirically on this dataset, the function seems to be *near-linear*. This is a massive reduction in the number of comparisons when compared to the quadratic number of comparisons shown earlier without any filtering.

Although both the filtering heuristics are independent of each other, they complement each other to effectively reduce a greater number of candidate comparisons together than alone.

The index data structure in conjunction with the above filtering heuristics form the key components of SourcererCC to achieve scalability. The next section describes the complete algorithm of SourcererCC.

3.4 Clone Detection Algorithm

The algorithm works in two stages: (i) Partial Index Creation; and (ii) Clone Detection. Each step has filtering heuristics directly embedded in it as described below.

3.4.1 Partial Index Creation

The basic approach of using Inverted index in SourcererCC is derived from the way keyword queries are answered using an inverted index in Information Retrieval []. An inverted index consists of a list of all the unique tokens that appear in any code block, and for each token, a list of the code blocks in which it appears. The index can be constructed in one sequential scan of the data. With the inverted index created, the query "find the code blocks where token X is present" can now be answered in a single random access.

SourcererCC's index creation step exploits Property 1.b and creates optimized indexes for tokens only in the sub-blocks. We call this *Partial Index*. In this Section, we first use an example to describe how SourcererCC creates partial index. Later we describe the formal algorithm for partial index creation.

To understand SourcererCC's partial index creation, consider the following four code blocks in the corpus:

$$B1 = [T7, T1, T3, T2, T8]$$
$$B2 = [T6, T7, T3, T4, T5]$$
$$B3 = [T1, T2, T5, T6, T7]$$
$$B4 = [T3, T2, T1, T7, T6]$$

Note that these code blocks are not ordered and hence we cannot compute their sub-blocks. Table 3.2 shows the occurrence frequency of each token in the corpus (column Global Occurrence Frequency). The tokens in the table are ranked in the ascending order of their occurrence frequency. As a result, less frequent tokens (e.g T4, T8) are ranked higher.

Token	Global Occurrence Frequency	Rank
T4	1	1
T8	1	2
T5	2	3
T1	3	4
T2	3	5
T3	3	6
T6	3	7
T7	4	8

Table 3.2: Global Token Frequency Map

The tokens in the four code blocks listed above are now re-arranged according to their rank in the Table 3.2. As a result, we get the following ordered code blocks:

$$B1 = [\underline{T8, T1}, T2, T3, T7]$$

$$B2 = [\underline{T4, T5}, T3, T6, T7]$$

$$B3 = [\underline{T5, T1}, T2, T6, T7]$$

$$B4 = [\underline{T1, T2}, T3, T6, T7]$$

Since the code blocks are now ordered, assuming θ is specified as 0.8, the sub-block (underlined tokens) of each code block can be calculated using Property 1.b. While creating the index, SourcererCC will index only the tokens in the sub-blocks (underlined tokens). As a result, creating $(1 - \theta) * 100\%$ smaller index as compared to a full index. Moreover, bigger code blocks lead to even greater reduction in the size of the partial index. This not only saves space but also enables faster retrieval because of a smaller index. In traditional index based approaches, all the tokens are indexed. However, as described later in Chapter 4, in our experiments $\theta = 0.7$ resulted in the best recall and precision, therefore, the index size is smaller by 70% as compared to a full index.

Algorithm 1 lists the steps to create a partial index. The first step is to iterate over each code block b ($line3$), and sort it according to the global token frequency map (GTP) ($line4$).

Algorithm 1 SourcererCC's Algorithm for Partial Index Creation

INPUT: B is a list of code blocks $\{b_1, b_2, ...b_n\}$ in a project/repository, GTP is the global token position map, and θ is the similarity threshold specified by the user
OUTPUT: Partial Index(I) of B

```
1: function createPartialIndex(B, θ)
2:     I = φ
3:     for each code block b in B do
4:         b = Sort(b, GTP)
5:         tokensToBeIndexed = |b| − ⌈ θ ·|b|⌉ + 1
6:         for i = 1 : tokensToBeIndexed do
7:             t = b[i]
8:             I_t = I_t ∪ (t, i)
9:         end for
10:    end for
11:    return I
12: end function
```

This is done as a prerequisite to the application sub-block overlap filtering. Next, the size of sub-block is computed using formula shown in Property 1.b, i.e., $(t - i + 1)$. Later, tokens in the sub-block are indexed to create a partial index. ($lines\, 6 - 8$).

3.4.2 Clone Detection

After partial index is created, the goal is to detect clones. Algorithm 2 describes the steps in detail. The *detectClones()* function iterates over each query block b, and sorts them using the same (GTP) that was created during index creation ($line\ 4$). Again, this is done as a prerequisite for both Property 1 & 2 to be applicable. After that, it calculates the length of query sub-block by using the same formula described in Property 1 ($line\ 5$). Next it iterates over only as many tokens as the length of b's sub-block and retrieves candidates by querying the partial index. Note that since partial index is created using only sub-blocks, the candidates retrieved in this phase implicitly satisfy Property 1. In other words, by creating partial index, the algorithm not only reduces the index size, but also ensures that we only get a filtered set of candidates that satisfy Property 1.

After the candidates are retrieved for a given query block, a trivial optimization to further eliminate candidates is done using the size of the candidates. That is, if a candidate c does not have enough tokens needed for it to be b's clone pair, then there is no point in even comparing them. This is done using a conditional check $|c| > \lceil\ \theta \cdot |b|\ \rceil$ on *line* 8. This further filters out false positive candidates.

The remaining candidates that have satisfied the above elimination process are now subjected to the filtering based on Property 2. First, based on θ, a threshold is computed to identify the minimum number of tokens needed to be matched for b and c to be considered as a clone pair (ct on *line* 9). Now, as the tokens in b and c are compared, a theoretical upper-bound is dynamically computed based on the number of remaining tokens in b and c (*line* 10). This upper-bound indicates, the maximum number of tokens b and c could match assuming all of their remaining tokens would match. If at any point in the iteration, the sum of upper-bound (i.e, the maximum number of tokens b and c could match) and the current similarity score (i.e, the number of tokens b and c have matched until now) happens to be less than ct (i.e, the minimum number of tokens b and c need to match), c is eliminated from b's candidate map, $candSimMap$ (*lines* 11 *and* 14). In other words, Property 2 is violated. On the other hand, if the sum is more than ct, the similarity between b and c gets updated with each matching token (*line* 12). Once all the tokens in b's sub-block are exhausted (*line* 19), we have a map of candidates ($candSimMap$) along with their similarity score and the last seen token in each candidate. The reason for storing the last seen token will become clear as we explain further. The next task is to verify if the candidates will eventually end up being b's clones. This is done in a call to *verifyCandidates()* function on *line* 18.

Algorithm 2 SourcererCC's Algorithm for Clone Detection

INPUT: B is a list of code blocks $\{b_1,\ b_2,...b_n\}$ in a project/repository, I is the partial index created from B, and θ is the similarity threshold specified by the user

OUTPUT: All clone classes ($cloneMap$)

```
1: function DETECTCLONES(B, I, θ)
2:     for each code block b in B do
3:         candSimMap = φ
4:         b = SORT(b, GTP)
5:         querySubBlock = |b| − ⌈ θ ·|b|⌉ + 1
6:         for i = 1 : querySubBlock do
7:             t = b[i]
8:             for each (c, j) ∈ I_t such that |c| > ⌈ θ · |b| ⌉ do
9:                 ct = ⌈ max(|c|, |b|) · θ ⌉
10:                uBound = 1 + min(|b| − i, |c| − j)
11:                if candSimMap[c] + uBound ≥ ct then
12:                    candSimMap[c] = candSimMap[c] + (1, j)
13:                else
14:                    candSimMap[c] = (0, 0) // eliminate c
15:                end if
16:            end for
17:        end for
18:        VERIFYCANDIDATES(b, candSimMap, ct)
19:    end for
20:    return cloneMap
21: end function
```

```
1: function VERIFYCANDIDATES(b, candSimMap, ct)
2:     for each c ∈ candSimMap such that candSimMap[c] > 0 do
3:         tokPos_c = Position of last token seen in c
4:         tokPos_b = Position of last token seen in b
5:         while tokPos_b < |b| && tokPos_c < |c| do
6:             if min(|b| − tokPos_b, |c| − tokPos_c) ≥ ct then
7:                 if b[tokPos_b] == c[tokPos_c] then
8:                     candSimMap[c] = candSimMap[c] + 1
9:                 else
10:                    if GTP[b[tokPos_b]] < GTP[c[tokPos_c]] then
11:                        tokPos_b + +
12:                    else
13:                        tokPos_c + +
14:                    end if
15:                end if
16:            else
17:                break
18:            end if
19:        end while
20:        if candSimMap[c] > ct then
21:            cloneMap[b] = cloneMap[b] ∪ c
22:        end if
23:    end for
24: end function
```

3.4.3 Candidate Verification

The goal of *verifyCandidates()* function three-fold. First, iterate over candidates c of query b that were not rejected in *detectClones()*, second, compute their similarity score with b, and third, either reject the candidates if the score does not meet the computed threshold ct or add them to the $cloneMap$ if it does.

In doing so, an important optimization is seen on *line* 5. Note that tokens are not iterated from the start but from the last token seen in b and c because earlier in *detectClones()* few tokens of b and c were already iterated to check if they satisfy Property 1 & 2 ($lines\ 6-8$). Hence the function avoids iterating over those tokens again. It is for this reason, in *detectClones()*, $candSimMap$ is designed to not only store candidates but also the last token seen in each candidate, i.e., $(Candidate, TokensSeenInCandidate)$ pair.

The rest of the function while iterating over the remaining tokens ensures that Property 2 holds in every iteration ($line\ 6$), and then increments the similarity score whenever there is a token match ($lines\ 7-8$). If in any iteration, Property 2 is violated, candidate is eliminated immediately without iterating over the remaining tokens ($line\ 17$). Thus further reducing the number of token comparisons.

Another trivial but important optimization is done while iterating over code blocks. Since b and c are already sorted using a global token frequency map (GTP), *verifyCandidates()* efficiently iterates over b and c by incrementing only the index of a block that has a lower globally ranked token ($lines\ 10-14$). Hence while iterating, except in the worst case when b & c happen to be clone pairs, time complexity is reduced from $O(|b| * |c|)$ to $O(|b| + |c|)$.

3.4.4 Revisiting the Research Questions

In this section, we revisit the research questions posed in the Introduction (Chapter 1) and discuss how SourcererCC's clone detection approach answers them.

Research Question 1. **[Design]** - How can we be more robust to the modifications in cloned code to detect near-miss Type-3 clones?

One of the distinguishing characteristics of SourcererCC compared to other token-based tools is its ability to detect near-miss Type-3 clones. The bag-of-tokens model and overlap similarity measure play an important role in detecting such clones. Type-3 clones are created by adding, removing, or modifying statements in a duplicated code fragment. Since the bag-of-tokens model is agnostic to relative token positions in the code block, it is resilient to such changes, and hence can detect near-miss clones as long as the code blocks share enough tokens to exceed a given overlap threshold.

Additionally, many Type-3 clones have modifications similar to swapping statements in code blocks, combining multiple condition expressions into one, changing operators in conditional statements, and the use of one language construct over the another (e.g. "for" vs. "while"). While these changes may exhibit semantic differences, they preserve enough syntactic similarity at a token level to be detected as similar. Detecting such clones can be difficult for other token-based approaches as they use token sequences as a unit of match [61]. While a token-sequence approach could merge nearby cloned sequences into Type-3 clones [40], they fail to detect the clones when the Type-3 gaps are too frequent or large.

Research Question 2. **[Computational Complexity]** - How can we reduce the $O(n^2)$ candidate comparisons to $O(c.n)$, where $c << n$?

The two filtering heuristics - sub-block overlap and token position filtering - along with the partial index, form the key components to reduce the number of candidate comparisons

in SourcererCC. The sub-block overlap filtering ensures that the code blocks that will be identified as clones share at least one token in their sub-blocks. The partial index uses this constraint to collect the list of candidates in $O(1)$ time by a simple index look-up. Since there are only few candidates that satisfy the filtering constraint, the overall candidate comparison is reduced to $O(c.n)$ from $O(n^2)$, where c ($<< n$) is the number of candidates that satisfy the filtering constraint.

The identified candidates are later verified if they meet the similarity threshold before being declared as clones. Token position filtering is used in this verification step to further eliminate candidates early on if their upper-bounds do not meet the similarity threshold.

Empirical evidence of the reduction in the number of candidate comparisons can be seen by revisiting the Figure 3.3. The points denoted by $\triangle$ show the number of candidate comparisons after applying the sub-block overlap filtering. The difference in the two curves formed by points denoted by $\triangle$ and $\circ$ shows the reduction in candidate comparisons before and after applying the filtering.

Research Question 3. **[Engineering]** - How can we make faster candidate comparisons without requiring much memory?

SourcererCC uses an inverted index to optimize for the speed of the clone detection process. While traditionally the index size can increase tremendously with the increase in the size of the dataset, the sub-block overlap filtering reduces the index size by constructing a partial index of only the first few tokens in each code block. As a result, the size of the index constructed is approximately 70% smaller than a full index. The partial index not only drastically reduces the memory footprint of SourcererCC, but it also significantly improves the retrieval process. In our experiments (described later in Chapter 4) to detect clones on a repository of 25,000 Java projects using SourcererCC, we found that the size of the partial index is only 1.2 GB. That is to say that the partial index can easily fit even in the memory

(RAM) of a standard workstation, and thus enables faster candidate comparisons without requiring much memory.

To summarize the answer to the research questions, it is not a single component but the harmony of many components, including the bag-of-tokens model, the filtering heuristics, and the index data structure, that come together to make SourcererCC effective in accurate, scalable, and fast code clone detection.

3.5 Implementation

This section provides implementation and execution details of SourcererCC. SourcererCC is hosted at GitHub and can be downloaded from `https://github.com/Mondego/SourcererCC`. SourcererCC is implemented as a Java library consisting of three main components: (i) Parser; (ii) Indexer; and (iii) Searcher. It is packaged as two jar files: (i) inputBuilderClassic.jar; and (ii) indexbased.SearchManager.jar. The *inputBuilderClassic.jar* is responsible for parsing operations whereas *indexbased.SearchManager.jar* is parameterized for performing index and search operations.

We describe the implementation of each of these components below.

3.5.1 Parser

The goal of the parser is to read the source files of a project or a repository and generate normalized code blocks in the form of bag-of-tokens. To understand the input and output of this parsing step, let us consider a Java source file that contains a method named *execute* in it (see Figure 3.5) as an input to the parser. The output of the parser is shown in Figure 3.6.

In the output file generated by the parser, each method is represented in a newline. Since

```
    * Execute all nestedTasks.
    */
    public void execute() throws BuildException {
          if (fileset == null || fileset.getDir(getProject())==null)
          {
                  throw new BuildException
                  ("Fileset was not configured");
          }
          for (Enumeration e = nestedTasks.elements();
          e.hasMoreElements();) {
                  Task nestedTask = (Task) e.nextElement();
                  nestedTask.perform();
          }
          nestedEcho.reconfigure();
          nestedEcho.perform();
    }
```

Figure 3.5: Sample code fragment as input to the parser

```
1@#@for@@::@@1, Fileset@@::@@1, perform@@::@@2, was@@::@@1,
configured@@::@@1, throw@@::@@1, if@@::@@1, elements@@::@@1,
null@@::@@2, nextElement@@::@@1, nestedTask@@::@@2, execute@@::@@1,
e@@::@@3, nestedTasks@@::@@1, throws@@::@@1, getDir@@::@@1,
void@@::@@1, Enumeration@@::@@1, nestedEcho@@::@@2, not@@::@@1,
new@@::@@1, getProject@@::@@1, fileset@@::@@2, hasMoreElements@@::@@1,
Task@@::@@2, public@@::@@1, reconfigure@@::@@1, BuildException@@::@@2
```

Figure 3.6: Output produced by the parser

```
1. @#@ (this only appears once at the start of the line)
2. , (read comma)
3.  @@::@@
```

Figure 3.7: Delimiters used in the output file produced by the parser

```
for@@::@@1
Fileset@@::@@1
perform@@::@@2
...
```

Figure 3.8: (Token, Frequency) pair representation in the output format

our input here has only one method, the output consists of only one line. There are a total of three delimiters used, and they appear in the order shown below (Figure 3.7).

To further understand the output format, let us split the output on the delimiter @#@. As a result, we get two strings i.e., LHS and RHS of @#@ delimiter. The LHS is a unique *blockId* that is automatically assigned to each method (or code block).

When we further split the RHS using the delimiter ',' (comma), we get a set of (token, frequency) pairs of the entire method. For example, in the above case, we would get the following strings after splitting on ',' (Figure 3.8).

The string for@@ :: @@1 implies that in *execute* method, token for appears once. Similarly, $perform$@@ :: @@2 implies that the token $perform$ is present twice. This can easily be obtained by splitting the (token, frequency) pair on delimiter @@ :: @@.

A *blockId* is a unique Id that identifies a line in the parsed output file. A block could be at any granularity level - file, method, code block, or even a statement. For the above example Id 1 uniquely identifies *execute*. It is important that these block Ids are unique positive numbers that are assigned to the code blocks in an increasing order. The increasing order of block ids is used to avoid comparing the same code blocks twice. For example if block 1 is compared with block 2, then because of their symmetry there is no need to compare block 2

with block 1. SourcererCC exploits increasing Ids to compare a given block with only blocks that have Ids greater than the given block. So in the case of block 1 & 2, block 1 will be compared to block 2 because 2 is greater than 1, but not vice-versa.

SourcererCC also reports clones using these blockIds. For example, if there are two cloned methods with blockId 31 and 89, SourcererCC will report them as clones using their blockIds separated with a , i.e., (31, 89). In order to be able to track the code blocks from blockIds, the parser also generates a bookkeeping file that maps blockIds to the path of the code block in the file system. The start and end line of the code block is also reported in the bookkeeping file.

The command to execute the parser is as follows:

```
>java -jar InputBuilderClassic.jar <src/path>
<path/code-blocks> <path/bookkeeping> functions java 0 0 10 0 8
```

There are 10 arguments controlling various parameters of the parser. They are as follows:
1: Path to the input source folder

2: Path where the parsed output code block file will be created

3: Path where the bookkeeping file (mapping ids to code blocks) will be created

4: Granularity at which input is to be parsed (e.g., functions or blocks)

5: Language of the source project(s) (cpp, java, c#, and c)

6: Minimum number of tokens to be considered in a block

7: Maximum number of tokens to be considered in a block

8: Minimum number of lines to be considered in a block

9: Maximum number of lines to be considered in a block

10: Number of threads to launch for parsing

Note that setting the minTokens/minLines (parameter 6 & 8) to zero means no minimum limit for the size of the code block, whereas setting maxTokens/maxLines (parameter 7 & 9) to zero means that there is no upper limit, i.e., code blocks of any size will be considered.

3.5.2 Indexer

After parsing the corpus, the Indexer applies filtering heuristics on the output produced by the parser and constructs a partial inverted index of the corpus according to the similarity threshold specified by the user.

The command to execute the Indexer is as follows:

```
java -jar indexbased.SearchManager.jar index 8
```

The indexbased.SearchManager.jar expects two arguments: (i) action: index/search; and (ii) similarity threshold: an integer between 1-10 (both inclusive).

The action could be either "index" or "search" depending on the task to be performed. The second argument, similarity threshold, instructs SourcererCC to detect clones that satisfy the given similarity threshold. For example, a similarity threshold of 8 indicates that the user wants to detect clone pairs that are at least 80% similar. Please note that the specified similarity threshold should be the same during indexing and searching. That is, if one is using a similarity threshold of 8 while indexing, then one should use the same similarity threshold while searching clones.

3.5.3 Searcher

Once the index is constructed, Searcher is launched to detect all the clones by querying the partial index. Searcher is also responsible for verifying the retrieved candidates based on the similarity threshold.

The command to execute the Searcher is as follows:

```
java -jar dist/indexbased.SearchManager.jar search 8
```

3.6 Chapter Summary

This chapter introduced SourcererCC, an accurate, near-miss clone detection tool that scales to several hundred million lines of code on a single standard machine. The core idea of SourcererCC is to build an optimized index of code blocks and compare them using a simple and fast bag-of-tokens strategy which is very effective in detecting near-miss clones. Several filtering heuristics are used to reduce the size of the index, which in turn, significantly reduce the number of code block comparisons to detect the clones. SourcererCC also exploits the ordering of tokens in a code block to estimate a live upper-bound on the similarity of code blocks in order to reject or accept a clone candidate with minimal token comparisons.

Chapter 4

Evaluation of SourcererCC

The evaluation of SourcererCC was partially conducted independently by Prof. Chanchal Roy and his Ph.D. student Jeffrey Svajlenko at University of Saskatchewan. Dr. Roy is a veteran in the field of code cloning and author of NiCad, a popular clone detector tool. He is also well known for his seminal work on the comparison and evaluation of code clone detection techniques and tools [107].

Part of the material in this chapter is included with the permission of the IEEE and based on our work in:

- Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.; Lopes, C. "SourcererCC: Scaling Code Clone Detection to Big-Code," International Conference on Software Engineering (ICSE'16), May 2016

In this chapter, we evaluate the execution and detection performance of SourcererCC. We begin by evaluating its execution time and scalability using subject inputs of varying sizes in terms of lines of code (LOC). We then demonstrate SourcererCC's execution for a large inter-project repository, one of the prime targets of scalable clone detection. We measure its

Tool	Scale/BigCloneBench	Mutation Framework
SourcererCC	Min length 6 lines, min similarity 70%, function granularity.	Min length 15 lines, min similarity 70%, function granularity.
CCFinderX	Min length 50 tokens, min token types 12.	Min length 50 tokens, min token types 12.
Deckard	Min length 50 tokens, 85% similarity, 2 token stride.	Min length 100 tokens, 85% similarity, 4 token stride.
iClones	Min length 50 tokens, min block 20 tokens.	Min length 100 tokens, min block 20 tokens.
NiCad	Min length 6 lines, blind identifier normalization, identifier abstraction, min 70% similarity.	Min length 15 lines, blind identifier normalization, identifier abstraction, min 70% similarity.

Table 4.1: Clone Detection Tool Configurations

clone recall using two benchmarks: The Mutation and Injection Framework [102, 119] and BigCloneBench [114, 118]. We measure the precision of our tool by manually validating a statistically significant sample of its output for the BigCloneBench experiment.

We compare SourcererCC's execution and detection performance against four publicly available clone detection tools, including CCFinderX [61], Deckard [55], iClones [40] and NiCad [28]. We include CCFinderX as it is a popular and successful tool, which has been used in many clone studies. We include Deckard, iClones and NiCad as popular examples of modern clone detection tools that support Type-3 clone detection. Moreover, recent benchmarking experiments suggest these tools to be state-of-the-art in the field [117, 118]. The selection criteria also included tools with best the scalability, recall, and/or most unique performance aspects for this study. We focus primarily on near-miss clone detectors, as Type-1 and Type-2 clones are relatively easy to detect. The configurations of these tools for the experiments are found in Table 4.1. These are targeted configurations for the benchmarks, and are based on extensive previous experiences of our collaborators with the tools [117, 118] as well as previous discussions with their developers, where available.

Our primary goal with SourcererCC is to provide a clone detection tool that scales efficiently

for large inter-project repositories with near-miss Type-3 detection capability. Most existing state-of-the-art tools have difficulty with such large inputs, and fail due to scalability limits [115, 116]. Common limits include untenable execution time, insufficient system memory, limitations in internal data-structures, or crashing or reporting an error due to their design not anticipating such a large input [115, 116]. We consider our tool successful if it can scale to a large inter-project repository without encountering these scalability constraints while maintaining a clone recall and detection precision comparable to the state-of-the-art. As our target we use IJaDataset 2.0 [3], a large inter-project Java repository containing 25,000 open-source projects (3 million source files, 250 MLOC) mined from SourceForge and Google Code.

4.1 Execution Time and Scalability

We evaluate the execution time and scalability of SourcererCC and compare it to the competing tools. Execution time primarily scales with the size of the input in terms of the number of lines of code (LOC) needed to be processed and searched by the tool. So this is the ideal input property to vary while evaluating execution performance and scalability. However, it is difficult to find subject systems that are large enough and conveniently dispersed in size. Additionally, a tool's execution time and memory requirements may also be dependent on the clone density, or other properties of the subject systems. It is difficult to control for these factors while measuring execution performance and scalability in terms of input size.

Our solution was to build inputs of varying convenient sizes by randomly selecting files from IJaDataset. This should ensure each input has similar clone density, and other properties that may affect execution time, except for the varying size in LOC. Each input has the properties of an inter-project repository, which is a target of large-scale clone detection. We created one input per order of magnitude from 1K LOC to 100 MLOC. We built the

LOC	SourcererCC	CCFinderX	Deckard	iClones	NiCad
1K	3s	3s	2s	1s	1s
10K	6s	4s	9s	1s	4s
100K	15s	21s	1m 34s	2s	21s
1M	1m 30s	2m 18s	1hr 12m 3s	MEMORY	4m 1s
10M	32m 11s	28m 51s	MEMORY	—	11hr 42m 47s
100M	1d 12h 54m s5s	3d 5hr 49m 11s	—	—	INTERNAL LIMIT

Table 4.2: Execution Time (or Failure Condition) for Varying Input Size

inputs such that each larger input contains the files of the smaller inputs. This ensures that each larger subset is a progression in terms of execution requirements. Lines of code were measured using cloc tool [2], and includes only lines containing code, and not comment or blank lines.

The execution time of the tools for these inputs is found in Table 4.2. The tools were executed for these inputs using the configurations listed under "Scale" in Table 4.1. The tools were executed on a machine with a 3.5 GHz quad-core i7 CPU, 12 GB of memory, a solid-state drive, and running Ubuntu 15.04. CCFinderX was executed on an equivalent machine running Windows 7. We use the same configurations for evaluating recall with BigCloneBench such that recall, execution performance, and scalability can be directly compared.

Scalability

SourcererCC is able to scale even to the largest input, with reasonable execution time given the input sizes. CCFinderX is the only competing tool to scale to 100MLOC, however it only detects Type-1 and Type-2 clones. The competing Type-3 tools encounter scalability limits before the 100MLOC input. Deckard and iClones run out of memory at the 100MLOC and 1MLOC inputs, respectively. NiCad is able to scale to the 10MLOC input, but refuses to execute clone detection on the 100MLOC input. In our previous experience [116], NiCad refuses to run on inputs that exceed its internal data-structure limits, which prevent executions that will take too long to complete. From our experiment, it is clear that the state-of-the-art Type-3 tools do not scale to large inputs, whereas SourcererCC can.

Execution Time

For the 1KLOC to 100KLOC inputs, SourcererCC has comparable execution time to the competing tools. iClones is the fastest, but it hits scalability issues (memory) as soon as the 1MLOC input. SourcererCC has comparable execution time to CCFinderX and NiCad for the 1MLOC input, but is much faster than Deckard. SourcererCC has comparable execution time to CCFinderX for the 10MLOC input size, but is much faster than NiCad. For the largest input size, SourcererCC is twice as fast as CCFinderX, although their execution times fall within the same order of magnitude. Before the 100MLOC input, SourcererCC and CCFinderX have comparable execution times.

SourcererCC is able to scale to inputs of at least 100MLOC. Its execution time is comparable or better than the competing tools. Of the examined tools, it is the only state-of-the-art Type-3 clone detector able to scale to 100MLOC. While CCFinderX can scale to 100MLOC for only detecting Type-1 and Type-2 clones, SourcererCC completes in half the execution time while also detecting Type-3 clones.

4.2 Experiment with Big IJaDataset

Since SourcererCC scaled to 100 MLOC without issue, we also executed for the entire IJaDataset (250MLOC). This represents the real use case of clone detection in a large inter-project software repository. We execute the tool on a standard workstation with a quad-core i7 CPU, 12GB of memory, and 100GB of SSD disk space. We executed SourcererCC using the "Scale" configuration in Table 4.1, with the exception of increasing the minimum clone size to ten lines. Six lines is common in benchmarking [13]. However, a six line minimum may cause an excessive number of clones to be detected in IJaDataset, and processing these clones for a research task can become another difficult scalability challenge [115]. Addition-

ally, larger clones may be more interesting since they capture a larger piece of logic, while smaller clones may be more spurious.

SourcererCC successfully completed its execution for IJaDataset in 4 days and 12 hours, detecting a total of 146 million clone pairs. The majority of this time was clone detection. Extracting and tokenizing the functions required 3.5 hours, while computing the global token frequency map and tokenizing the blocks required only 20 minutes. SourcererCC required 8GB of disk space for its pre-processing, index (1.2GB), and output. Of the 4.7 million functions in IJaDataset greater than 10 lines in length, 2.4 million (51%) appeared in at least one clone pair detected by SourcererCC. We have demonstrated that SourcererCC scales to large inter-project repositories on a single machine with good execution time. We have also shown that building an index is an inexpensive way to scale clone detection and reduce overall execution time.

Since CCFinderX scales to the 100MLOC sample, we also executed it for IJaDataset. We used the same settings as the scalability experiment. We did not increase CCFinderX's minimum clone size from 50 tokens, which is roughly 10 lines (assuming 5 tokens per line). This was not an issue with benchmarking as we used a 50 token minimum for reference clones from BigCloneBench. CCFinderX executed for 2 days before crashing due to insufficient disk space. Its pre-processed source files (25GB) and temporary disk space usage exceeded the disk space. Based on the findings of a previous study, where CCFinder was distributed over a cluster of computers [83], we can estimate it would require 10s of days to complete detection on 250MLOC, given sufficiently large disk-space. So we can confidently say that SourcererCC is able to complete quicker, while also detecting Type-3 clones.

4.3 Recall

In this section we measure the recall of SourcererCC and the competing tools. Recall has been very difficult for tool developers to measure as it requires knowledge of the clones that exist in a software system [104, 103]. Manually inspecting a system for clones is non-trivial. Even a small system like Cook, when considering only function clones, has almost a million function pairs to inspect [123]. Bellon et al. [13] created a benchmark by validating clones reported by the clone detectors themselves. This has been shown to be unreliable for modern clone detectors [117]. Updating this benchmark to evaluate a tool would require extensive manual clone validation with a number of modern tools. As such, many clone detection tool papers simply do not report recall.

In response, our collaborators created The Mutation and Injection Framework [102, 119], a synthetic benchmark that evaluates a tool's recall for thousands of fine-grained artificial clones in a mutation-analysis procedure. The framework is fully automatic, and requires no validation efforts by the tool developer. However, we recognized that a modern benchmark of real clones is also required. So we also use BigCloneBench [114], a recently developed benchmark containing 8 million validated clones based on clone functionality within and between 25,000 open-source projects. It measures recall for an extensive variety of real clones produced by real developers. The benchmark was designed to support the emerging large-scale clone detection tools, which previously lacked a benchmark. This combination of real-world and synthetic benchmarking provides a comprehensive view of SourcererCC's clone recall.

4.3.1 Recall Measured by the Mutation Framework

The Mutation Framework in a synthetic benchmark that evaluates tools using artificial constructed clones. The advantage of this benchmark is it can precisely evaluate the detection capabilities of the tools at a fine granularity.

The Mutation Framework follows a standard mutation-analysis procedure. It begins by extracting a randomly selected code fragment (a function or a block) from a repository of varied source code. A copy of the fragment is mutated using one of fifteen mutation operators, which are listed in Table 4.3. Each mutation operator performs a single code edit, and is based on a empirically validated taxonomy of the types of edits developers make on copy and pasted code. The original and mutant versions of the code fragment are randomly injected into a copy of a subject system, evolving the system by a single copy, paste and modify clone. The clone detector under evaluation is executed for the mutant subject system, and its recall of the injected clone is measured using a clone-matching algorithm. This process is repeated many times per mutation operator, and the clone detector's recall can be measured per type of edit developers make on cloned code. A more detailed overview of this methodology can be found in the literature [119, 102].

4.3.1.1 Procedure

We executed the framework for three programming languages: Java, C and C#, using the following configuration. For each language, we set the framework to generate clones using 250 randomly selected functions, 10 randomly selected injection locations, and the 15 mutation operators, for a total of 37,500 unique clones per language (112,500 total). For Java we used JDK6 and Apache Commons as our source repository and IPScanner as our subject system. For C we used the Linux Kernel as our repository and Monit as our subject system. For C# we use Mono and MonoDevelop as our repository, and MonoOSC as our subject system.

ID	Mutation (Edit) Description	Clone Type
mCW_A	Addition of whitespace.	Type-1
mCW_R	Removal of whitespace.	
mCC_BT	Change in between token (/**/) comments.	
mCC_BT	Change in end of line (//) comments.	
mCF_A	Change in formatting (add newline).	
mCF_R	Change in formatting (remove newline).	
mSRI	Systematic renaming of an identifier.	Type-2
mARI	Renaming of a single identifier instance.	
mRL_N	Change in value of a numeric literal.	
mRL_S	Change in value of a string literal.	
mSIL	Small insertion within a line.	Type-3
mSDL	Small deletion within a line.	
mIL	Insertion of a line.	
mDL	Deletion of a line.	
mML	Modification of a line.	

Table 4.3: Cloning Mutation Operators

We constrained the synthesized clones to the following properties: (1) 15-200 lines in length, (2) 100-2000 tokens in length, and (3) a mutation containment of 15%. We have found this configuration provides accurate recall measurement [117, 118]. The tools were executed and evaluated automatically by the framework using the configurations listed in Table 4.1. To successfully detect a reference (injected) clone, a tool must report a candidate clone that subsumes 70% of the reference clone by line, and appropriately handles the clone-type specific edit introduced by the mutation operator [119].

4.3.1.2 Results

Recall measured by the Mutation Framework for SourcererCC and the competing tools is summarized in Table 4.4. The table shows recall per clone type. SourcererCC has perfect recall for the first three clone types, including the most difficult Type-3 clones, for Java, C and C# indicating that SourcererCC's clone detection algorithm is capable of handling all the types of edits developers make on copy and pasted code for these languages, as outlined

Tool	Java			C			C#		
	T1	T2	T3	T1	T2	T3	T1	T2	T3
SourcererCC	100	100	100	100	100	100	100	100	100
CCFinderX	99	70	0	100	77	0	100	78	0
Deckard	39	39	37	73	72	69	-	-	-
iClones	100	92	96	99	96	99	-	-	-
NiCad	100	100	100	99	99	99	98	98	98

Table 4.4: Mutation Framework Recall Results

in the editing taxonomy for cloning [108].

SourcererCC exceeds the competing tools with the Mutation Framework. The runner up is NiCad, which has perfect recall for Java, and near-perfect recall for C and C#. iClones is also competitive with SourcererCC, although iClones has some troubles with a small number of Type-2 and Type-3 clones. SourcererCC performs much better for Type-2 and Type-3 clones than CCFinderX. Of course, as a Type-2 tool, CCFinderX does not support Type-3 detection. SourcererCC performs much better then Deckard across all clone types. While Deckard has decent recall for the C clones, its Java recall is very poor. We believe this is due to its older Java parser (Java-1.4 only), while the Java reference clones may contain up to Java-1.6 features.

In summary, SourcererCC has perfect recall with the Mutation Framework, which shows it can handle all the types of edits developers make on cloned code. As per standard mutation analysis, the Mutation Framework only uses one mutation operator per clone. This allows it to measure recall very precisely per type of edit and clone type. It also prevents the code from diverging too far away from natural programming. However, this means that the Mutation Framework makes simple clones. It does not produce complex clones with multiple type of edits, and the Type-3 clones it produces generally have a higher degree of syntactical similarity. To overcome this issue, we use the real-world benchmark BigCloneBench as follows.

4.3.2 Recall Measured by BigCloneBench

Here we measure the recall of SourcererCC using BigCloneBench and compare it to the competing tools. We evaluate how its capabilities shown by the Mutation Framework translate to recall for real clones produced by real developers in real software-systems, spanning the entire range of clone types and syntactical similarities. Together the benchmarks provide a complete view of SourcererCC's recall.

BigCloneBench [114] is a large clone benchmark of manually validated clone pairs in the inter-project software repository IJaDataset 2.0 [3]. IJaDataset consists of 25,000 open-source Java systems spanning 3 million files and 250MLOC. This benchmark was built by mining IJaDataset for functions implementing particular functionalities. Each clone pair is semantically similar (by their target functionality) and is one of the four primary clone types (by their syntactical similarity). The published version of the benchmark considers 10 target functionalities [114]. For this study, we use an in-progress snapshot of the benchmark with 48 target functionalities, and 8 million validated clone pairs.

For this experiment, we consider all clones in BigCloneBench that are 6 lines or 50 tokens in length or greater. This is a standard minimum clone size for benchmarking [13, 118]. The number of clones in BigCloneBench, given this size constraint, is summarized per clone type in Table 4.5. There is no agreement on when a clone is no longer syntactically similar, so it is difficult to separate the Type-3 and Type-4 clones in BigCloneBench. Instead the authors of the BigCloneBench divide the Type-3 and Type-4 clones into four categories based on their syntactical similarity, as follows. Very Strongly Type-3 (VST3) clones have a syntactical similarity between 90% (inclusive) and 100% (exclusive), Strongly Type-3 (ST3) in 70-90%, Moderately Type-3 (MT3) in 50-70%, and Weakly Type-3/Type-4 (WT3/4) in 0-50%. Syntactical similarity is measured by line and by token after Type-1 and Type-2 normalizations. The categories, and the benchmark in general, are explained in more detail

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4
# of Clone Pairs	35787	4573	4156	14997	79756	7729291

Table 4.5: BigCloneBench Clone Summary

in the literature [114].

4.3.2.1 Procedure

We executed the tools for IJaDataset and evaluated their recall with BigCloneBench. As we saw previously (Section 4.1), most tools do not scale to the order of magnitude of IJaDataset (250MLOC). Our goal here is to measure recall not scalability. We avoid the scalability issue by executing the tools for a reduction of IJaDataset with only those files containing the known true and false clones in BigCloneBench (50,532 files, 10MLOC). Some of the competing tools have difficulty even with the reduction, in which case we partition it into small sets, and execute the tool for each pair of partitions. In either case, the tool is exposed to every reference clone in BigCloneBench, and it is also exposed to a number of false positives as well, thus creating a realistic input. We measure recall using a subsume-based clone-matching algorithm with a 70% threshold. A tool successfully detects a reference clone if it reports a candidate clone that subsumes 70% of the reference clone by line. This is the same algorithm used in Mutation Framework, and is standard in benchmarking [13].

4.3.2.2 Results

Recall measured by BigCloneBench is summarized in Table 4.6. Recall is summarized per clone type and per Type-3/4 category for all clones, as well as specifically for the intra and inter-project clones.

SourcererCC has perfect detection of the Type-1 clones in BigCloneBench. It also has near-

Tool	All Clones						Intra-Project Clones						Inter-Project Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4	T1	T2	VST3	ST3	MT3	WT3/T4	T1	T2	VST3	ST3	MT3	WT3/T4
SourcererCC	100	98	93	61	5	0	100	99	99	86	14	0	100	97	86	48	5	0
CCFinderX	100	93	62	15	1	0	100	89	70	10	4	1	98	94	53	1	1	0
Deckard	60	58	62	31	12	1	59	60	76	31	12	1	64	58	46	30	12	1
iClones	100	82	82	24	0	0	100	57	84	33	2	0	100	86	78	20	0	0
NiCad	100	100	100	95	1	0	100	100	100	99	6	0	100	100	100	93	1	0

Table 4.6: BigCloneBench Recall Measurements

perfect Type-2 detection, with a negligible difference between intra and inter-project clones. This shows that the 70% threshold is sufficient to detect the Type-2 clones in practice. SourcererCC has excellent Type-3 recall for the VST3 category, both in the general case (93%) and for intra-project clones (99%). The VST3 recall is still good for the inter-project clones (86%), but it is a little weaker. SourcererCC's Type-3 recall begins to drop off for the ST3 recall (61%). Its recall is good in this Type-3 category for the intra-project clones (86%) but poor for the inter-project clones (48%). We believe this is due to Type-3 clones from different systems having a higher incidence of Type-2 differences, so the inter-project clones in the ST3 category are not exceeding SourcererCC's 70% threshold. Remember that the reference clone categorization is done using syntactical similarity measured after Type-2 normalizations, whereas SourcererCC does not normalize the identifier token names (to maintain precision and index efficiency). Lowering SourcererCC's threshold would allow these to be detected, but could harm precision. SourcererCC has poor recall for the MT3 and WT3/T4, which is expected as these clones fall outside the range of syntactical clone detectors [118]. Of course, Type-4 detection is outside the scope of this study.

Compared to the competing tools, SourcererCC has the second best recall overall, with NiCad taking the lead. Both tools have perfect Type-1 recall, and they have similar Type-2 recall, with NiCad taking a small lead. SourcererCC has competitive VST3 recall, but loses out in the inter-project case to NiCad. SourcererCC is competitive with NiCad for intra-project clones in the ST3 category, but falls significantly behind for the inter-project case and overall. NiCad owes its exceptional Type-3 recall to its powerful source normalization capabilities.

However, as we saw previously in Section 4.1, NiCad has much poorer execution time for larger inputs, and hits scalability constraints at the 100MLOC input. So SourcererCC instead competes with execution performance and scalability, making these tools complimentary tools for different use cases.

Comparison to CCFinderX is interesting as it is the only other tool to scale to the 100MLOC input. Both tools have comparable Type-1 and Type-2 recall, with SourcererCC having the advantage of also detecting Type-3 clones, the most difficult type. While BigCloneBench is measuring a non-negligible VST3 recall for CCFinderX, it is not truly detecting the Type-3 clones. As shown by the Mutation Framework in Table 4.4, CCFinderX has no recall for clones with Type-3 edits, while SourcererCC has perfect recall. Rather, CCFinderX is detecting significant Type-1/2 regions in these (very-strongly similar) Type-3 clones that satisfy the 70% coverage threshold. This is a known limitation in real-world benchmarking [117, 118], which is why both real-world and synthetic benchmarking is needed. CCFinderX's detection of these regions in the VST3 is not as useful to users as they need to manually recognize the missing Type-3 features. CCFinderX's Type-3 recall drops off past the VST3 category, where Type-3 gaps are more frequent in the clones. While we showed previously that CCFinderX also scales to larger inputs (Section 4.1), SourcererCC's faster execution, Type-3 support, and better recall make it an ideal choice for large-scale clone detection.

Deckard and iClones are the other competing Type-3 clone detectors. Both SourcererCC and iClones have perfect Type-1 recall, but SourcererCC exceeds iClones in both Type-2 and Type-3 detection, and iClones does not scale well. Deckard has poor overall recall for all clone types, along with its scalability issues.

4.4 Precision

In this section we measure SourcererCCs precision and compare it against the competing tools. Unlike clone detection recall, where there exists benchmarks [117], measuring precision remains an open problem, and there is no standard benchmark or methodology. Instead we estimate the precision of the tools by manually validating a random sample of their output, which is the typical approach.

From each tool we randomly selected 400 of the clone pairs they detected in the recall experiment. The validation efforts were equally distributed over five reviewers, all software researchers with 2-5 years of software development experience, with each validating 80 clones from each tool. The clones were shuffled and the reviewers were kept blind of the source of each clone. The reviewers were familiar with the cloning definitions, and were asked to validate the clones as per their judgment.

Clone validation can be very subjective, so we used multiple reviewers to get an average opinion of what is a true positive clone. In addition to the 400 clones, we had each reviewer validate the same sample of 100 clones (20 from each tool) to measure agreement in the validation. Agreement between pairs of reviewers for these clones ranged from 67% to 90%, with an average of 80%. All five reviewers agreed on 58% of the clones, and at least 4 reviewers agreed on 86% of the clones. Overall, the reviewers had strong agreement; suggesting that the reviewers have a similar opinion on what constitutes a true positive clone. Disagreement is expected due to the subjective nature of true vs false positive clones. By distributing the clones amongst multiple reviewers we normalize for different opinions on what constitutes a true positive clone.

We find that SourcererCC has a precision of 83%, the second best precision of these tools. This is a very strong precision as per the literature [103, 108, 104], and demonstrates the accuracy and trustworthiness of SourcererCC's output. We summarize the precision of all the

	SourcererCC	CCFinderX	Deckard	iClones	NiCad
Precision	83	72	28	91	56
Precision (min 10 LOC)	86	79	30	93	80
Recall[1]	90	75	53	78	99
F-Measure[2]	88	77	38	85	88

Table 4.7: Tool Recall and Precision Summary

[1] Including T1, T2, VST3 and ST3.
[2] F-Measure = 2 * (recall * precision) / (recall + precision)

tools in Table 4.7, and contrast it against their overall recall measured by BigCloneBench. We do not include the MT3 and WT3/T4 clones as they are outside the scope of these tools. iClones has the top precision (91%) because it is cautious when reporting Type-3 clones, although this results in a Type-3 recall (38%) significantly below the other tools. SourcererCC's bag-of-tokens model and similarity threshold allows it to provide a good balance of recall and precision while also providing superior scalability. NiCad has a precision of 56%, possibly because of its use of normalizations and relaxed threshold. However, with these settings NiCad also has a very strong recall (99%). CCFinderX's precision, while competitive, is low considering it only targets Type-1 and Type-2 clones (although it detects some Type-1/2 regions in Type-3 clones). Deckard has very poor precision in this experiment, reporting some clones that are very dissimilar. This may be because we relaxed its similarity threshold to detect more Type-3 clones. The authors [55] report a precision of 94% for Java-1.4 code with a 100% similarity threshold. Nonetheless, CCFinderX and Deckard show very poor Type-3 recall as well.

Tool configuration, particularly minimum clone size, could be a potential bias in this precision experiment. This was controlled in the recall experiment by setting a minimum clone size of six lines and 50 tokens in BigCloneBench, and configuring the tools appropriately. However, there is no agreement between lines of code and the tokens contained, and even the tools measure lines (original/pretty-printed) and tokens (original/filtered) in different ways. This makes comparing the precision of the tools difficult because this configuration issue may

cause a tool to detect many small spurious clones that another tool does not detect due to difference in clone size configuration and/or measurement. To examine this, we re-measured precision using a minimum clone size of 10 original lines of code in order to harmonize the minimum clone size of the tools. We used the existing validation efforts, randomly selecting 30 validated clones per tool per judge (150 clones per tool) that are 10 LOC or greater. These results are shown in Table 4.7. All of the tools see a boost in precision, although NiCad most significantly. This implies that with full normalization and a generous threshold of 30% dissimilarity, NiCad may be detecting small false clones that are 6 (pretty-printed) lines or so, but contain very few tokens (spurious similarity).

4.5 Summary of Recall and Precision Experiments

Figure 4.1 summarizes the results of recall and precision experiments for all the tools. It reports recall computed using BigCloneBench and Mutation Framework, precision, and F-Measure computed using precision and recall from BigCloneBench. Note that for CCFinder, recall for Mutation Framework is computed only for Type-1 and Type-2 clones as CCFinder does not support Type-3 clones. NiCad demonstrates the highest recall whereas iClones demonstrates the highest precision. While SourcererCC is not number one in recall or precision, it provides the best balance of both measures (as seen from F-Measure), while achieving high scalability and good execution performance. SourcererCC owes its high recall performance to its bag-of-tokens approach, while maintaining high precision by not using heavy normalizations on the tokens.

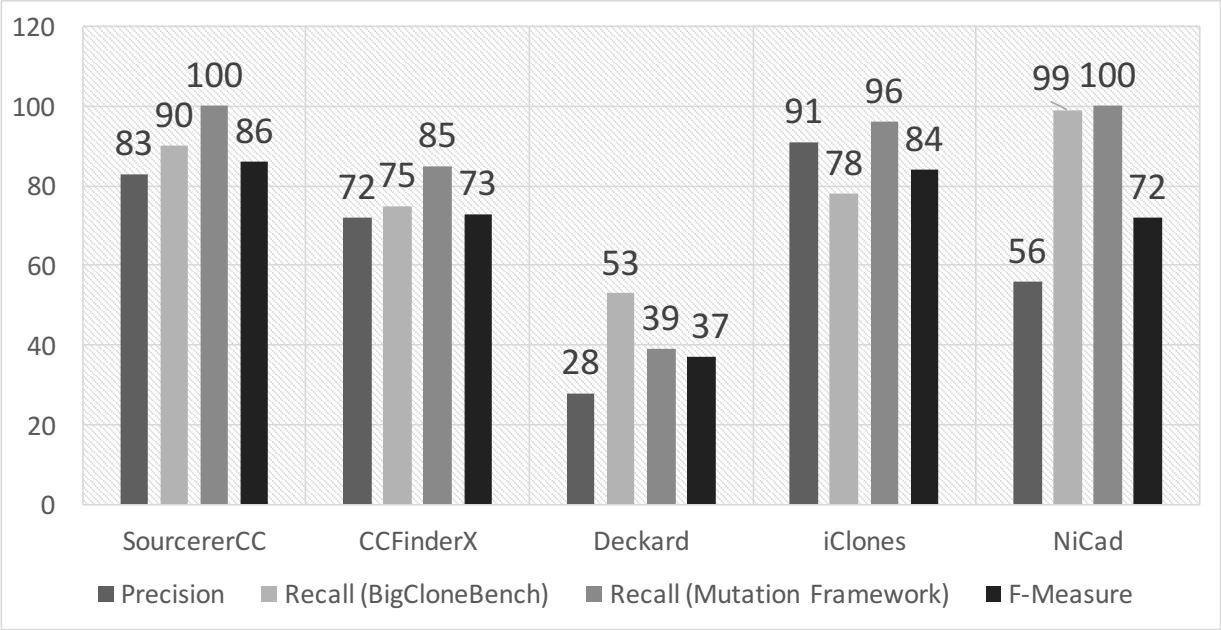

Figure 4.1: Summary of Results. F-Measure is computed using Recall (BigCloneBench) and Precision

4.6 Sensitivity Analysis of the Similarity Threshold Parameter

The goal of this experiment is to study the impact of change in the similarity threshold value on three metrics: (i) the number of clones detected; (ii) the total number of candidates detected; and (iii) the total number of tokens compared.

Subject Systems. We chose 35 open source Apache Java projects as subject systems for this experiment. These projects are of varied size and span across various domains including search and database systems, server systems, distributed systems, machine learning and natural language processing libraries, network systems, etc. Most of these subject systems are highly popular in their respective domain. Such subject systems exhibiting variety in size and domain help counter a potential bias of our study towards any specific kind of software system. More details about the size of the projects (LOC) and number of methods present in them can be found in Appendix A.

SourcererCC was used to detect clones individually for each project, with varying values of similarity threshold (θ) from 0.6 to 0.9. Next, for each value of θ, the above mentioned metrics were measured.

We observed all the metric values to decrease with the increase in the value of θ. We posit the following two reasons for the observed phenomenon:

First, as we increase θ, the size of sub-block (calculated using Sub-block overlap filtering) to validate candidacy of each code block decreases. Second, a higher value of θ makes the Token Position filter stricter and therefore fewer number of code block satisfy the similarity constraint eventually decreasing the number of candidates. In summary, the two filtering heuristics work together to ensure the decrease in the number of candidates with the increase in the values of θ. Moreover, since the number of tokens compared is directly related to the number of candidates compared. The total number of tokens compared also decline with the increase in the value of θ,

Higher values of θ also mean that the code blocks should contain a greater number of similar tokens to be identified as clones. As a result, SourcererCC finds near exact clones (e.g. Type-1 & Type-2) when configured with higher values of θ. Since projects usually have fewer number of Type-1 & Type-2 clones, we expect the total number of clones reported in a project to decrease with increase in the value of θ.

Figure 4.2 validates these observations visually on five projects by plotting the impact of change in the value of θ value on: (i) the number of clones reported; (ii) the total number of candidates compared; and (iii) the total number of tokens compared. The X-axis shows θ increasing from 0.6 to 0.95 with a step size of 0.05. The Y-axis shows the number of clones reported (top-center), the number of candidates compared (bottom-left), and number of token compared (bottom-right).

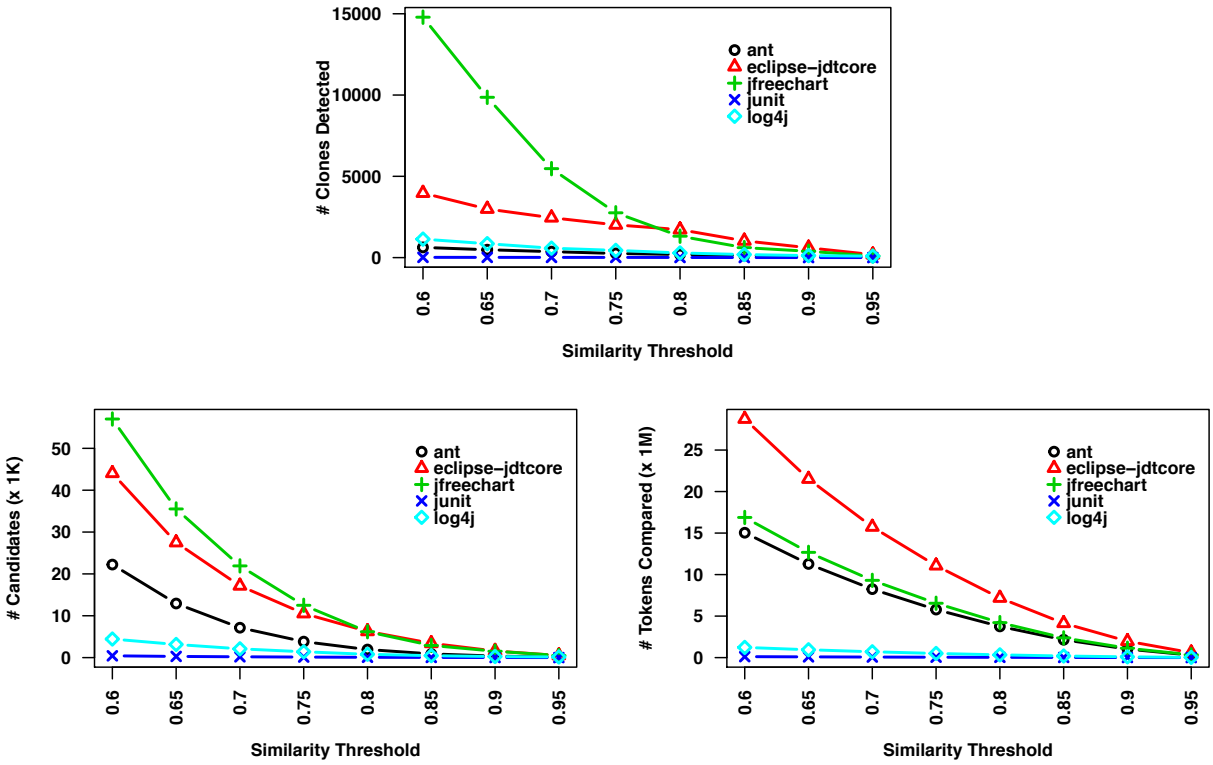

Figure 4.2: Change in number of clones reported (top-center), number of candidates compared (bottom-left), and number of tokens compared (bottom-right) with the change in similarity threshold.

Subject System	Similarity Threshold= (θ)											
	$\theta = 6$			$\theta = 7$			$\theta = 8$			$\theta = 9$		
	Clones	Candidates	Tokens	Clones	Candidates	Tokens	Clones	Candidates	Tokens	Clones	Candidates	Tokens
ant	627	22,227	15,039,736	367	7,144	8,246,641	192	1,932	3,757,357	70	365	1,027,302
berkeleyparser	1,898	23,945	6,748,359	893	9,049	3,739,054	453	2,973	1,731,236	185	758	495,463
cglib	96	2,345	628,941	48	963	347,875	16	331	160,055	3	53	43,126
cloud9	4,668	49,093	11,438,967	3,141	20,967	6,310,099	2,074	8,329	2,882,845	893	3,044	797,171
cocoon	64	952	265,473	29	359	147,462	16	128	66,769	5	26	18,257
commons-io	190	1,449	283,116	111	697	161,750	53	332	78,415	18	87	22,145
dom4j	546	4,399	775,743	385	1,993	444,833	211	867	211,605	62	240	59,027
eclipse-ant	121	3,158	1,077,231	80	1,166	588,075	34	345	270,151	12	70	74,001
eclipse-jdtcore	3,970	44,051	28,738,282	2,455	17,150	15,739,712	1,720	6,272	7,183,914	604	1,580	1,976,385
hadoop-hdfs	9,797	38,667	16,416,789	6,620	21,561	9,058,565	3,782	10,374	4,138,223	1,885	4,430	1,136,957
hadoop-mapred	633	11,902	9,378,871	392	4,532	5,136,312	216	1,531	2,343,023	71	394	638,329
hibernate	703	23,278	12,137,476	393	7,900	6,677,260	205	2,399	3,051,344	76	499	821,415
j2sdk1.4.0-javax-swing	2,852	50,474	39,610,627	1,439	17,926	22,058,818	621	5,409	10,214,080	240	1,111	2,835,348
jfreechart	14,790	57,012	16,882,529	5,473	21,913	9,306,027	1,317	6,187	4,215,650	391	1,526	1,133,525
junit	16	428	115,762	13	193	67,331	8	73	32,413	3	17	8,867
jython	5,634	58,439	28,411,746	3,878	22,174	15,692,490	2,843	7,759	7,221,636	1,798	2,872	2,031,600
log4j	1,136	4,441	1,230,706	580	2,101	703,410	286	795	333,128	124	260	92,428
lucene	126	2,136	621,783	76	909	348,860	56	331	162,262	24	92	45,243
mahout-core	832	16,914	9,213,897	350	6,162	4,985,376	181	1,915	2,256,816	68	418	619,650
mason	1,543	7,418	2,673,334	968	3,459	1,483,762	494	1,508	683,485	212	527	193,088
netbeans-javadoc	121	712	252,375	93	356	144,082	59	166	65,989	31	69	18,103
nutch	62	1,204	456,655	43	484	253,037	26	179	114,899	10	52	31,143
pdfbox	60	758	187,242	37	309	103,349	31	109	49,223	15	38	14,490
pig	3,262	33,661	15,250,182	1,799	12,313	8,237,661	1,179	4,416	3,714,378	557	1,552	1,027,363
pmd	3,787	13,900	6,369,775	1,755	6,079	3,586,033	528	1,965	1,663,434	93	305	452,392
poi	4,553	15,059	4,073,584	2,557	6,670	2,309,619	1,129	2,540	1,082,874	400	622	295,137
postgresql	728	4,250	1,324,880	390	1,778	733,598	251	757	336,378	122	233	93,395
rhino	670	14,381	5,834,893	326	5,067	3,219,519	129	1,640	1,475,011	27	329	395,847
stanford-nlp	2,088	67,973	46,547,595	1,078	21,852	25,669,854	586	5,892	11,845,562	214	1,177	3,326,717
struts	306	4,002	1,183,743	199	1,551	647,443	132	577	293,699	97	183	80,984
substance	469	7,355	3,564,768	297	2,866	1,927,020	198	1,113	867,580	74	331	237,918
synapse-core	567	7,097	3,513,705	346	2,792	1,907,701	166	1,022	862,906	15	267	231,192
tomcat-catalina	1,142	19,196	9,568,188	669	7,205	5,233,239	329	2,289	2,377,121	166	577	652,348
uima-core	186	1,082	342,724	101	504	193,919	62	269	90,564	23	80	25,659
xerces	1,137	14,342	7,910,217	692	5,563	4,386,906	398	1,985	2,037,094	160	555	562,347

Table 4.8: Impact of change in the similarity threshold value on: (i) the number of clones detected; (ii) the total number of candidates; and (iii) the total number of tokens compared.

Table 4.8 shows the complete set of results for all the projects for θ ranging from 0.6 to 0.9. For each value of θ, the table shows the number of clones reported (sub-column Clones), the number of candidates compared (sub-column Candidates), and the number of tokens compared (sub-column Tokens). The data confirms the decreasing trend for all the columns with the increase in the value of θ.

4.7 Manual Inspection of Clones Detected by SourcererCC

In order to gain more insights about what kind of clones are detected by SourcererCC, two reviewers (software engineering researchers with 2-5 years of software development experience) manually inspected 198 clone groups found in the above listed 35 Java projects. The number of clone siblings in these clone groups ranges from 2 to 5,473. The reviewers' classification of these clone groups coincided with the categories proposed by Kasper and Godfrey for classification in [63]. The following are various categories that the reviewers classified the clone groups in.

Cross-cutting Concerns. Two different methods responsible for different tasks can look like clones when, apart from implementing the required behavior, they also implement the code for addressing similar aspects (e.g. logging, authentication, and debugging). In such cases, the code that implements the actual logic in a method is small. However, oftentimes developers copy the whole method body from another method and change only the part of the code that affects the functionality - leaving the two methods structurally very similar. Since a large part of such methods contain code that makes function calls to logging, debugging and authentication modules, it often is a well tested code - making such clones harmless.

The code snippets 1A and 1B in Figure 4.3 show one such clone group where the methods

use *HttpServletRequest* API. Both the methods perform user authentication before making a function call to the *putAttribute* method. The presence of authentication code in both the methods makes the code structurally very similar.

Code Generation. The methods that are auto-generated using some tool usually have similar structures. Such clones are auto-generated from specifications and ideally not meant to be modified by the developers. As long as this code of conduct is followed, presence of such clones is not a threat.

As an example of this type of cloning, we found a clone group with 5,473 members in *PyArrayDerived.java* in *Jython* project. The code snippet 2 shown in Figure 4.3 is written for *rlshift* operator and is duplicated for all the operators creating clones. Similarly, we found clones of *initComponents()* method in *FormEditor.java* file of *Eclipse-jdt core* project which gets regenerated by the constructor of *FormEditor*.

API/Library Protocols. We found clones resulting from the constraints or style imposed by the frameworks. Cloning in such case is because of the limited APIs made available by the framework and the sequence in which they are to be invoked to achieve a specific task. Such tasks are usually domain independent and can also occur across projects (e.g. adding items in the menu, calendar functionality, internationalization, among others). For example, we found a clone of *createActionComponent()* (see the code snippet 3 in Figure 4.3), a Factory method which creates the *JMenuItem* for each *Action* added to the *JMenu*. Similarly the *run* method in *edu.umd.cloud9.example.hits.AFormatterWG* and *edu.umd.cloud9.example.hits.HFormatterWG* has a series of similar function calls to create and configure the mapper, reducer, and combiner for invoking map-reduce jobs.

Replicate and Specialize. While developing a solution to a problem, the developers sometime copy a piece of code that has already been implemented to address a similar problem, and make some modifications to address the required problem.

```
1A) public void processNestedTag(PutTag nestedTag) throws JspException {
    // Check role
    HttpServletRequest request = (HttpServletRequest)
    pageContext.getRequest();
    String role = nestedTag.getRole();
    if (role != null && !request.isUserInRole(role)) {
        // not allowed : skip attribute
        return;
    }
    putAttribute(nestedTag.getName(), nestedTag.getRealValue());
}
/*********************************************/

1B) public void putAttribute(PutListTag nestedTag) throws JspException {
    // Check role
    HttpServletRequest request = (HttpServletRequest)
    pageContext.getRequest();
    String role = nestedTag.getRole();
    if (role != null && !request.isUserInRole(role)) {
        // not allowed : skip attribute
        return;
    }
    putAttribute(nestedTag.getName(), nestedTag.getList());
}
/*********************************************/

2)  public PyObject __rlshift__(PyObject other) {
    PyType self_type = getType();
    PyObject impl = self_type.lookup("__rlshift__");
    if (impl != null) {
        PyObject res = impl.__get__(this, self_type).__call__(other);
        if (res == Py.NotImplemented)
            return null;
        return res;
    }
    return super.__rlshift__(other);
}
/*********************************************/
```

```
3)  protected JMenuItem createActionComponent(Action a) {
    JMenuItem mi = new JMenuItem((String) a.getValue(Action.NAME),
            (Icon) a.getValue(Action.SMALL_ICON)) {
            protected PropertyChangeListener
            createActionPropertyChangeListener(Action a) {
            PropertyChangeListener pcl = createActionChangeListener(this);
            if (pcl == null) {
                pcl = super.createActionPropertyChangeListener(a);
            }
            return pcl;
        }
    };
    mi.setHorizontalTextPosition(JButton.TRAILING);
    mi.setVerticalTextPosition(JButton.CENTER);
    mi.setEnabled(a.isEnabled());
    return mi;
}
/*********************************************/

4A) public boolean containsAll(final IntList c) {
    boolean rval = true;
    if (this != c) {
        for (int j = 0; rval && (j < c._limit); j++) {
            if (!contains(c._array[j])) {
                rval = false;
            }
        }
    }
    return rval;
}
/*********************************************/

4B) public boolean removeAll(final IntList c) {
    boolean rval = false;
    for (int j = 0; j < c._limit; j++) {
        if (removeValue(c._array[j])) {
            rval = true;
        }
    }
    return rval;
}
```

```
5A) public void startEntity(String name) throws SAXException {
    for (int i = 0; i < this.lexicalHandlerList.length; i++) {
        if (this.lexicalHandlerList[i] != null)
            this.lexicalHandlerList[i].startEntity(name);
    }
}
/*********************************************/

5B) public void endEntity(String name) throws SAXException {
    for (int i = 0; i < this.lexicalHandlerList.length; i++) {
        if (this.lexicalHandlerList[i] != null)
            this.lexicalHandlerList[i].endEntity(name);
    }
}
/*********************************************/

6A) public static string getAuthority(String uri) {
    RE re = RE(uripattern);
    if(re.match(uri)) {
        return re.getParen(4);
    } else {
        throw new illegalArgumentException(uri + "␣is␣not␣a␣correct␣URI");
    }
}
/*********************************************/

6B) public static string getPath(String uri) {
    RE re = RE(uripattern);
    if(re.match(uri)) {
        return re.getParen(5);
    } else {
        throw new illegalArgumentException(uri + "is␣not␣a␣correct␣URI");
    }
}
```

Figure 4.3: Sample code clones observed in the subject systems. 1A & 1B: Cross-cutting Concerns; 2: Code Generation; 3: API/Library Protocols; 4A, 4B & 5A, 5B: Replicate and Specialize; 6A & 6B: Near-Exact Copy

We noticed several cases where the clone siblings perform exactly the opposite tasks, but are structurally very similar. For example, the code snippets 4A and 4B in Figure 4.3 show two methods that iterate over all the elements of a user defined list and invoke *contains* and *removeValue* APIs respectively. Undoubtedly, these two code blocks share most of the code; however, their existence in two separate well defined methods could be a thoughtful decision. Similarly, the code snippets 5A and 5B in Figure 4.3 show another such example of two methods which share most of the code, but are designed to achieve separation of concerns.

Exact or Near-Exact Copy. We found several instances where the clone siblings have identical or near identical code. Such cases of clones could arise due to laziness on programmer's part or because of bad development practices. In some cases we noticed comments by the developers reflecting their awareness of the presence of such clones, and their intent to refactor in the future. For example, we found that the two siblings of a clone group had identical code except hard coded integer values passed as parameters to the methods (6A & 6B in Figure 4.3). Although this does not result in any serious threat to the program, it is definitely a bad practice because it breaks abstraction and increases the code size. An easy fix is to refactor the two methods into one by modifying the signature of any of these methods to accept an integer type as a parameter. A tool support to pro-actively detect such patterns can help a developer fix the code before cloning. There exists automated refactoring support for such scenarios; however, such an approach is reactive, and hence often ends up as an optional exercise. We conjecture that a more proactive approach of making a developer aware of such issues while she is copy pasting will blend in with the development activity and hence will be more easily adopted.

Overall, in all of the observed clone groups, we came to an understanding that the developers seemed to have copied code, which is sufficiently well written and handle the edge cases. Cases where cloning also resulted into duplication of bad practices and hence impacted overall

code quality, could have been fixed by adding proactive tool support to help developers make an informed decision. In their study [113], Thummalapenta et al. also found that clones were consistently propagated when needed and developers actually seem to remember the clone locations that require such propagation, particularly in important cases like bug fixes. This suggests that although there may be many inconsistent changes, developers are aware and make conscious decision about propagating the change.

4.8 Threats to Validity

As observed by Wang et al. [124], clone detection studies are affected by the configurations of the tools, and SourcererCC is no exception. However, we carefully experimented with its configurations to achieve an optimal result. As for the other tools, we also conducted test experiments, and also discussed with the corresponding developers for obtaining proper configurations, where available. Their configurations also provided good results in the past studies [118, 117, 115].

There are some limitations in the precision measurement. The choice of subject system (in our case a subset of IJaDataset), tool configuration [124], and targeted use case [123] can all have a significant impact on the precision measured. The reliability and experience of reviewers is also identified as a concern [5, 19, 123]. To combat this, we split the validation efforts across five reviewers so that the measurement reflects the views of multiple clone researchers. Moreover, conducting this in a blind manner ensured no subjective bias towards any specific tool.

4.9 Chapter Summary

In this chapter, we evaluated SourcererCC's performance and detection quality. We demonstrated SourcererCC's scalability with IJaDataset, a large inter-project repository containing 25,000 open-source Java projects, and more than 250 MLOC. In our experiments, with four state-of-the-art tools, we found that SourcererCC is the only tool to scale to the complete repository. We measured recall using two state-of-the-art clone benchmarks, the Mutation Framework, and BigCloneBench, and found that SourcererCC is competitive with even the best of the state-of-the-art Type-3 clone detectors. We measured precision by five reviewers, manually inspecting statistically significant sample of tools' output, and found SourcererCC to also have high precision (86%). These results suggests that SourcererCC can be an ideal choice for the various modern use cases that require reliable, complete, fast, and scalable clone detection.

Chapter 5

SourcererCC-D: Parallel and Distributed SourcererCC

Part of the material in this chapter is included with the permission of John Wiley and Sons and based on our work in:

- Sajnani, H.; Saini, V.; Lopes, C., "A Parallel and Efficient Approach to Large Scale Clone Detection," Journal of Software: Evolution and Process (JSEP), June 2015 vol., no. 27, pp. 402-429, doi: 10.1002/smr.1707

5.1 Introduction

SourcererCC advances the state-of-the-art in code clone detection tools that can scale vertically using high power CPUs and memory added to a single machine. While this approach works well in most of the cases, in certain scenarios using vertical scalable approaches may not be feasible as they are bounded by the amount of data that can fit into the memory of a

single machine. In such scenarios, timely computing clones in ultra large datasets is beyond the capacity of a single machine.

Under such scenarios, efficient parallelization of the computation process is the only feasible option. Previously, research in this direction was limited due to the lack of the availability of resources and the cost of setting up the infrastructure. But the recent developments in the field of cloud computing and the availability of low-cost infrastructure services like Amazon Web Services (AWS), Azure, and Google Cloud, have enabled the research in this area.

However, it is important to note that simply dividing the input space and parallelizing the clone detection operation do not solve the problem, because running tools on projects individually, and then combining the results in the later step, would lead to a collection of common clones, but would not identify clones across division boundaries. Additionally, oftentimes, simple distributed approaches consume substantial time in aggregating the information after the clone detection step. For example, Livieri et al. [83] proposed D-CCFinder, an in-house distributed CCFinder to detect Type-2 clones. It took two days to analyze 400 million lines of code with a cluster of 80 machines, suggesting that while input partitioning can scale existing non-scalable detectors, it significantly increases the cumulative runtime. Thus, efficient parallelization of the computation process is necessary. Moreover, to the best of our knowledge, currently there is no[1] distributed clone detector available.

This chapter presents SourcererCC-D, a distributed and parallel version of SourcererCC that is designed to horizontally scale to multiple processors and efficiently detect first three types of clones on large datasets. Since SourcererCC forms the core engine of SourcererCC-D, it has enhanced scalability (horizontal) preserving the same detection quality (recall and precision) as SourcererCC.

The rest of the chapter describes SourcererCC-D's architecture, followed by its implementa-

[1]As per the interaction with the author of D-CCFinder, the tool was developed only for the experimentation and is no longer available.

tion details and experiments to measure its scalability using AWS clusters.

5.2 Architecture

SourcererCC-D detects clones on a cluster of nodes. The core idea is to first construct the index of the entire corpus that is shared across all the nodes, and parallelize the clone searching process by distributing the tasks across all the nodes in the cluster. In order to achieve this, SourcererCC-D can follow a standard *Shared-disk* (see Figure 5.1) or a *Shared-memory* (see Figure 5.2) architecture style.

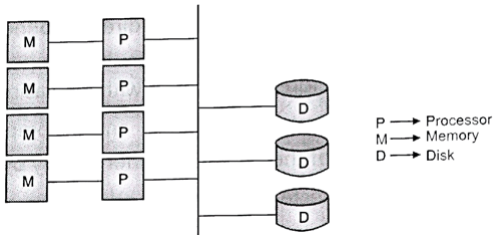

Figure 5.1: Shared-disk Architectural Style

A shared disk architecture (SD) is a distributed computing paradigm in which all disks are accessible from all the cluster nodes. While the nodes may or may not have their own private memory, it is imperative that they at least share the disk space. A shared memory architecture (SM) is a distributed computing paradigm in which the cluster nodes not only share the disks but also have access to a global shared memory.

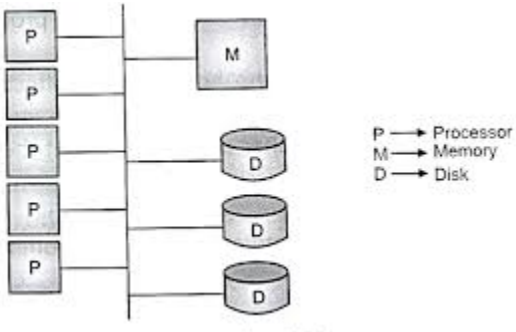

Figure 5.2: Shared-memory Architectural Style

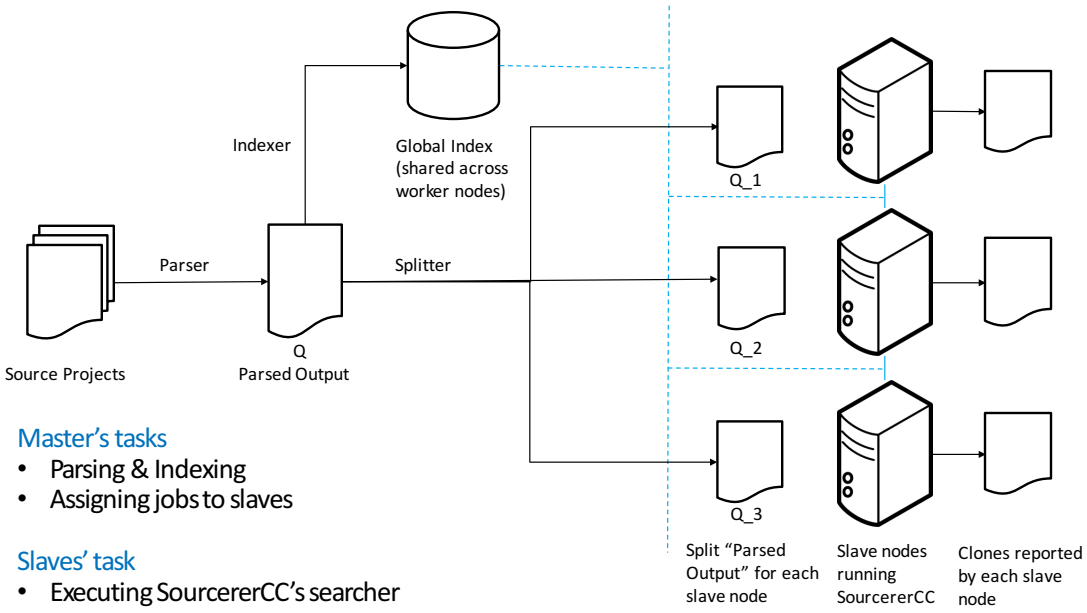

Figure 5.3: SourcererCC-D's Clone Detection Process

Figure 5.3 describes SourcererCC-D's clone detection process. Let us assume a cluster of $N + 1$ nodes[2]. Initially, a master node (any one of the cluster nodes) runs the parser on the entire corpus and produces a parsed input file containing code blocks according to the granularity set by the user. Next, the master node runs SourcererCC's Indexer on the parsed input file to build an index. The constructed index, also known as a global index, resides on the shared disk, and hence is accessible by all the nodes in the cluster. After the global index is constructed, the master node splits the parsed input file into N different files namely Q_1, Q_2, Q_3,...Q_N and distributes them to each node in the cluster. Each node is now responsible to locally compute clones of code blocks in its respective query file using SourcererCC's Searcher. Note that since each node has access to the global index, it can find all the clones in the entire corpus for its given input, i.e., clones present across other nodes. It is for this reason, nodes must have a shared disk space to store the global index. When all the nodes finish executing the Searcher, all the clones in the corpus have been found.

Note that in the above design, while the search phase is distributed and happens in parallel,

[2]In case of a single high performance multi-processor machine, $N+1$ is the number of processors available on that machine

the index construction phase is not parallelized (only the master node constructs the index). However, this hardly impacts the overall clone detection performance because we found that index construction takes less than 4% of the total time to detect clones.

SourcererCC-D can be deployed on in-house clusters, cloud services like AWS, or even multi-processor machines. Appendix B describes how to run it on AWS.

5.3 Evaluation

5.3.1 Evaluation Metrics

SourcererCC-D is built on top of SourcererCC. Hence, it bears same recall and precision as SourcererCC. However, since SourcererCC-D's primary goal is to achieve horizontal scalability, we conduct several experiments to measure its performance in terms of two important properties: *Speed-up* and *Scale-up*. Speed-up and Scale-up are two well-known metrics to measure the performance of parallel systems. They are described as follows:

Speed-up is the extent to which more hardware can perform the same task in less time than the original system. With added hardware, speed-up holds the task constant and measures time savings. For example, Figure 5.4 shows how each parallel hardware system performs half of the original task in half the time required to perform it on a single system. With good speed-up, additional processors reduce system response time. Speed-up is measured using this formula:

$$\text{Speed-up} = \frac{\text{Time\_Original}}{\text{Time\_Parallel}} \tag{5.1}$$

where $Time\_Parallel$ is the elapsed time spent by a larger, parallel system on the given task.

For example, if the original system took 60 seconds to detect clones in a project, and two parallel systems took 30 seconds, then the value of speed-up would equal to two.

Thus a speed-up value of n, where n times more hardware is used, indicates the ideal or linear speed-up. In other words, an ideal speed-up is when twice as much hardware can perform the same task in half the time (or when three times as much hardware performs the same task in a third of the time, and so on).

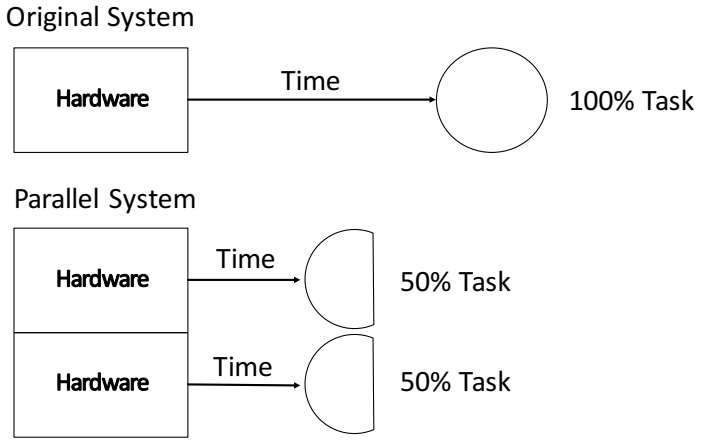

Figure 5.4: Speed-up

Scale-up is the factor m that expresses how much more work can be done in the same time period by a system n times larger. With added hardware, a formula for scale-up holds the time constant, and measures the increased size of the job which can be done. With good scale-up, if the dataset grows, you can keep response time near constant by adding CPUs.

Scale-up is measured using this formula:

$$\text{Scale-up} = \frac{\text{Task\_Parallel}}{\text{Task\_Original}} \tag{5.2}$$

where $Task\_Parallel$ is the task processed in a given amount of time on a parallel system.

For example as shown in Figure 5.5, if the original system can detect clones of 100 code blocks in a given amount of time, and the parallel system can detect clones of 200 code blocks in

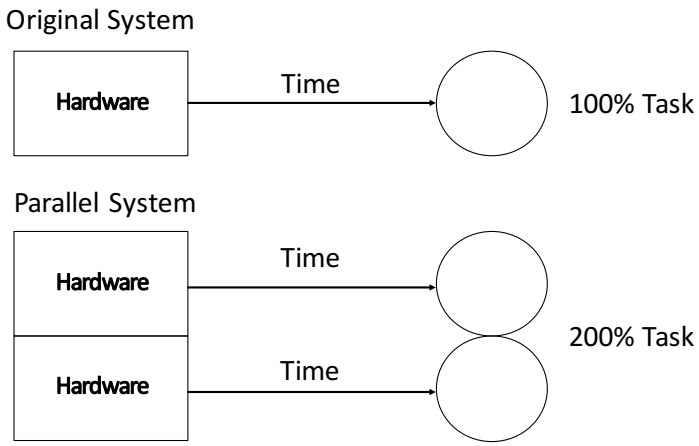

Figure 5.5: Scale-up

the same amount of time, then the value of scale-up would be equal to two (i.e., 200/100 = 2). A scale-up value of two indicates the ideal scale-up when twice as much hardware can process twice the data volume in the same amount of time. However, in a real life scenario it is difficult to measure how much increase in data volume would be appropriate for a given increase in the hardware.

5.3.2 Experiments to Measure the Speed-up

In order to measure the speed-up of SourcererCC-D, we set up five clusters of 1, 2, 4, 8 and 16 nodes respectively using AWS. Each node in the cluster has the following specification: High Frequency Intel Xeon E5-2680 v2 (Ivy Bridge) Processors, 7.5 GB memory, 80 GB storage, 4 virtual core and Ubuntu 10.2 Operating System.

The dataset consists of source code files totaling 100 MLOC and 3,484,913 methods. This is a subset of SourcererCC's evaluation dataset from IJaDataset repository.

We compute all the clones in the dataset separately using each cluster. The idea is to examine the change in computation time with the increase in cluster size. Table 5.1 shows the speed-up results. Column 1 (Cluster Size) represents the number of nodes in the cluster,

Column 2 (Clone Detection Time) shows the time taken to compute clones, and Column 3 (Clone Pairs Detected) shows the total number of clone pairs detected. Note that the total clone pairs detected is the same for all the runs as clone detection is on the same dataset (indicating that there is no information loss in parallelization). Row 1 shows the time taken to compute clones when there is no parallelization, (i.e., Cluster Size of 1), and subsequent rows show the decrease in time as the cluster size increases. On examining closely, it appears that whenever the cluster size doubles, the clone detection time is reduced approximately by a factor of two, exhibiting almost linear speed-up of two. This is also shown in graph plotted in Figure 5.6.

Cluster Size	Clone Detection Time	Clone Pairs Detected
1	40h:06m:35s	55,431,324
2	20h:06m:01s	55,431,324
4	10h:08m:32s	55,431,324
8	05h:01m:27s	55,431,324
16	02h:33m:24s	55,431,324

Table 5.1: Speed-up results

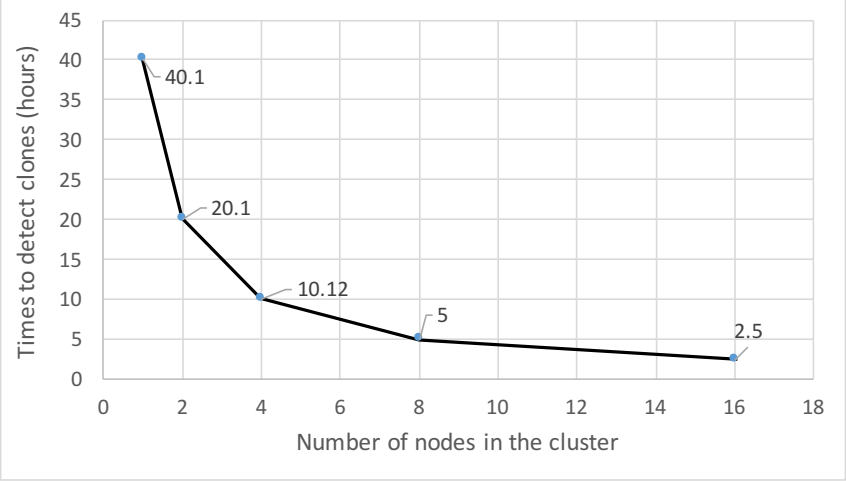

Figure 5.6: Speed-up

Cluster Size	Dataset Size (# methods)	Clone Detection Time	Clone Pairs Detected
1	174,246	00h:21m:50s	2,428,043
2	348,492	00h:21m:29s	2,765,863
4	696,983	00h:30m:15s	4,243,682
8	1,393,966	00h:46m:32s	10,557,301
16	2,787,931	01h:40m:08s	35,936,960

Table 5.2: Scale-up results

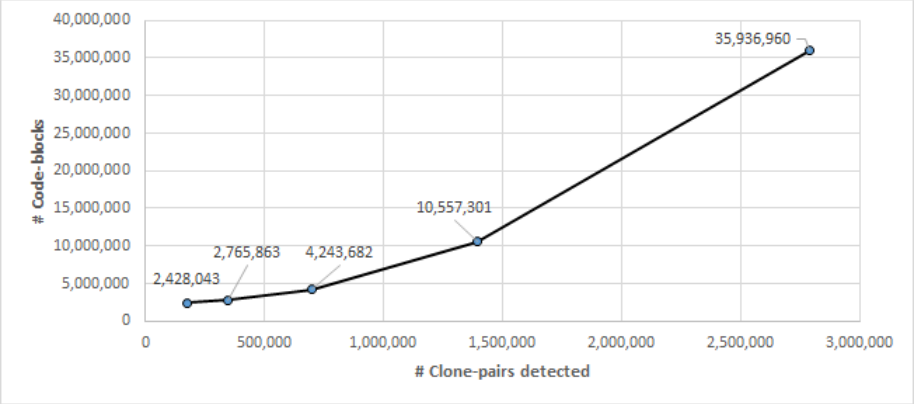

Figure 5.7: The number of clone-pairs detected increases exponentially with the increase in number of code blocks

5.3.3 Experiments to Measure the Scale-up

For measuring the scale-up, we again set up five clusters with the same node configuration as above. However, this time we also increase the size of the dataset proportionately as we increase the number of nodes in the cluster.

Table 5.2 shows the time taken to detect clones (Column 3) and the number of clone-pairs detected (Column 4) as the number of methods increases (Column 2). As shown in the table, the cluster size and dataset are increased in the same proportion. That is, as the cluster-size is doubled (Column 1), the number of methods in the dataset is also doubled to measure if the time taken to detect clones holds constant. However, it is observed that as the number of methods increases, the time taken to detect clones also increases despite a proportionate increase in the number of nodes. This is because (i) not all methods have an equal number

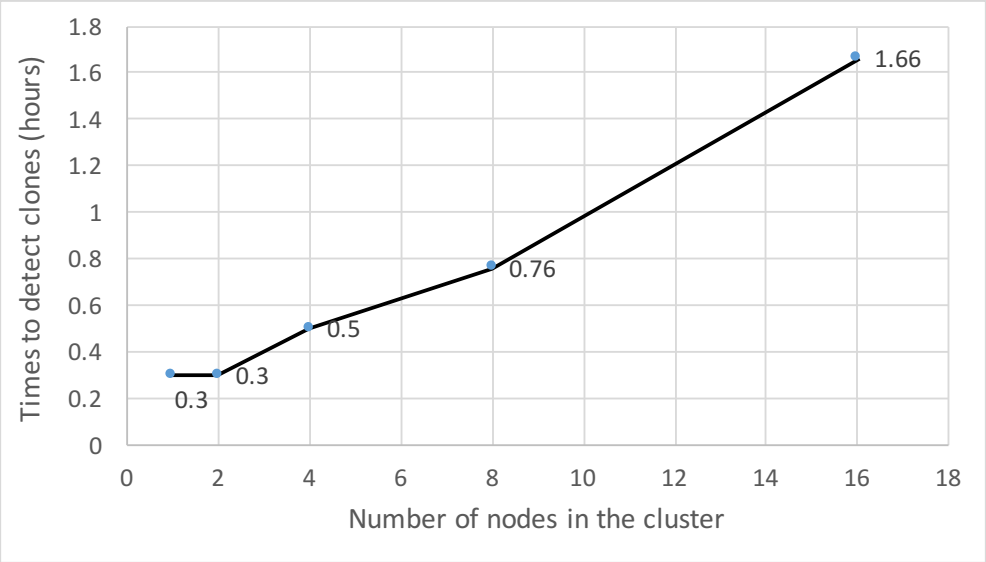

Figure 5.8: Scale-up

of clones, so the work done to detect clones for the added methods might be more than the earlier methods; and (ii) the number of clone pairs increases exponentially with the size of the dataset. That is, the bigger the dataset, the bigger the probability of finding clones. This is observed from Column 2 and Column 4 of the Table 5.2. Also, Figure 5.7 visually confirms this by plotting a chart of the number of clone pairs (Column 4) vs. the number of code-blocks (Column 2). Due to the above reasons, the time taken to detect clones is not constant but rather increases near *linearly* with the increase in the dataset as shown in Figure 5.8.

5.3.4 Detecting Project Clones in the MUSE Repository

5.3.4.1 DARPA's MUSE Initiative

The U.S. Defense Advanced Research Projects Agency is attempting to take big data analytics to the next level through a big code project designed to improve overall software reliability through a large-scale repository of software that drives big data. The DARPA big

code initiative, formally known as Mining and Understanding Software Enclaves (MUSE), seeks to leverage software analysis and big data analytics to improve the way software is built, debugged, and verified. The goal of the big code effort is to apply the principles of big data analytics to identify and understand deep commonalities among the constantly evolving body of software drawn from the hundreds of billions of lines of open source code available today [30].

5.3.4.2 The MUSE Repository

The MUSE repository consists of 151,135 Java projects containing 12,168,632 Java files. These projects are collected from various repositories including GitHub, Apache, Maven, Google Code, and SourceForge. These projects are of varying sizes and span across various domains including search and database systems, server systems, distributed systems, machine learning and natural language processing libraries, and network systems. Figure 5.9 shows the size distribution of projects in the repository. The X-axis represents the number of Java files (log scale), and the Y-axis represents the number of projects. Similarly Figure 5.10 shows the size distribution of projects where X axis represents lines of code on a log scale.

5.3.4.3 Heuristic to Detect Project Clones in the MUSE Repository

We wanted to use SourcererCC-D to detect all the project clones in the MUSE repository. To achieve this goal, we begin by first detecting all the file-level clones in the repository. File level clones are computed at 80% similarity threshold, i.e., two files are clones of each other if they share at least 80% of the tokens present in them. Next, we derive project clones from these computed file-level clones using the following heuristic: If a project A has at least 80% of its files reported as clones in some another project B, then A is considered to be a clone of project B. This is easily achieved by grouping computed file-level clones by their respective

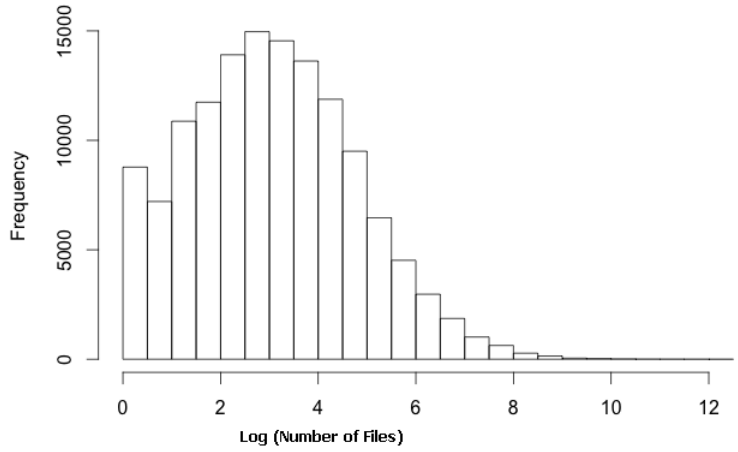

Figure 5.9: Size distribution of projects. Size is defined as Number of Files

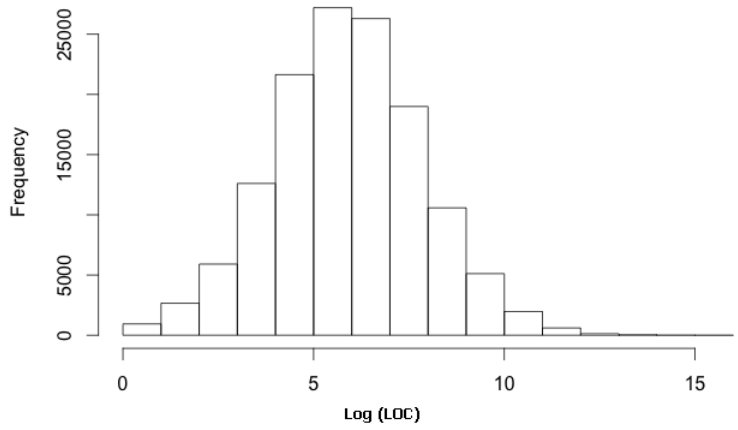

Figure 5.10: Size distribution of projects. Size is defined as LOC

projects.

5.3.4.4 Running SourcererCC-D on MUSE repository

The MUSE repository resides on a high performance multi-processor computing machine with the following hardware specification: Architecture: x86-64, # CPU(s): 40, Threads per core: 2, Model name: Intel(R) Xeon(R) CPU E5-2690 v2 3.00 GHz, CPU MHz: 2391.210, Memory: 125 GB, and Storage: 877 GB.

The first task is to run the parser at a file-level granularity on the entire corpus of 151,135 projects and generate a parsed input file. Next, we split this parsed input into 10 chunks and launch SourcererCC-D on the above machine with 10 CPUs as the worker nodes. Each instance is alloted a 10 GB of memory for its execution. It took 03h:49m to create the Index and 6d:16h:12m for all the nodes to detect all the file-level clones in the entire corpus. The task of aggregating these file-level clones to derive project clones was performed outside the SourcererCC-D's infrastructure.

5.3.4.5 Results

We summarize our key findings from the analysis of output produced by SourcererCC-D on the MUSE repository below.

(i) There exists 155,781,688 ($\approx$ 156 million) file-level clone pairs in the MUSE repository.

(ii) After aggregating these file-level clones to derive project clones, we found that 50,032 out of 151,135 projects are cloned at least once (33.10%).

(iii) 21,850 (14%) projects are cloned in two or more projects.

(iv) 23,544 (16%) projects contain two or more project clones in the repository.

(v) 38,537 (25.5%) projects are symmetrical clones at 80% similarity threshold. A symmetrical project clone pair at 80% similarity threshold means that at least 80% of the files in both the projects are common.

(vi) 30,783 (20.37%) projects are symmetrical clones at 100% similarity threshold, i.e., exact duplicate projects.

A qualitative manual inspection of hundreds of random clone pairs by the people involved in the project showed no false positives.

The large amount of cloning in this corpus was a surprising finding that was analyzed in greater depth by people involved in that project. Part of the reason for so much duplication was attributed to the fact that this corpus includes projects from different public repositories, some of which lost popularity (e.g. SourceForge) while others gained popularity (e.g. Github) over the years. It appears that many projects have presence in many of these public repositories. But even when taking that into account, this study found a considerable amount of code duplication within repositories, ranging from 7% to 18%.

This experiment successfully demonstrates the ability of SourcererCC-D to scale to very large datasets on high performance computing machines. To the best of our knowledge, this is the first experiment to detect clones at such a large scale.

5.4 Chapter Summary

Research in software clones lends itself easily to industrial application. However, scalability and impractical runtime are major issues for industrial adoption of existing techniques and tools. This chapter presented SourcererCC-D, a practical and inexpensive technique that uses a cluster of commodity machines for large scale code clone detection by exploiting the

inherent parallelism present in the problem. Our experiments demonstrated SourcererCC-D's ability to achieve ideal speed-up and near linear scale-up on large datasets.

This chapter also demonstrated that SourcererCC-D can be easily deployed on both AWS and high performance multi-processor machines. Using cloud services like AWS as a deployment environment provides advantages like load balancing, data replication, and fault tolerance over any other in-house distributed solutions where these things are to be dealt with explicitly.

Chapter 6

SourcererCC-I: Interactive SourcererCC for Developers

Part of the material in this chapter is included with the permission of the IEEE and based on our work in:

- Saini, V.; Sajnani, H.; Kim, J.; Lopes, C., "SourcererCC and SourcererCC-I: Tools to Detect Clones in Batch mode and During Software Development," International Conference on Software Engineering (ICSE'16), May 2016

6.1 Introduction

Over the past decade, several techniques and tools have been proposed for detecting code clones. However, as Lee et al. [80] point out, most clone detectors take a "post-mortem" approach involving the detection of clones "after" code development is complete. While such stand-alone tools [41, 105, 62] are beneficial in the analysis and investigation of code clones

and their evolution, they fail to provide necessary clone management support for clone-aware development activities, as they are not integrated with the development environment [129]. We carried out a preliminary survey to assess the need for a clone detection tool that is integrated with the software development environment among developers and received a compelling positive (90%) response (see Section 6.2). Also, Lague et al. [78] conducted a case study to assess the impact of integrating clone detection with the development process as a preventive control for maintenance issues. They analyzed 89 million lines of code and found several opportunities where the integration of automated clone detection with the development process could have helped.

Moreover the few clone detection tools that are integrated with the development environments are mostly focused on detecting Type-1 and Type-2 clones, and typically report all the clones in the entire code base. Such flooding of information may overwhelm the developer, who in practice, is likely to be interested in only the clones of a certain portion of code she deals with at a time.

To address these issues, we developed SourcererCC-I, an Eclipse plug-in based on top of SourcererCC, that instantaneously reports intra- and inter-project method level clones in Java projects. SourcererCC-I identifies the method a developer is currently working on, and then pro-actively reports the clones of that method in a non-obtrusive manner. The developer can decide to refactor the identified clones or invoke existing code instead of introducing a new clone. An instant clone search feature during development can also be useful in finding API usage examples or similar code fragments from curated repositories to facilitate code reuse.

6.2 A Preliminary Survey

As part of this project, we conducted a preliminary survey to understand how software practitioners perceive the use of clone detection tools during software development and maintenance activities. The survey consisted of 72 participants, out of which 66 had at least one year of industrial software development experience at the time of completing our survey. Figure 6.1 describes the participants' experience profile in detail.

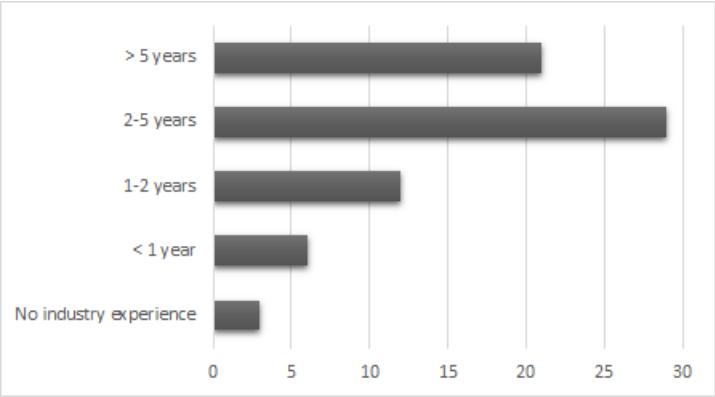

Figure 6.1: Industrial Experience of Survey Participants

The survey was designed with the following questions:

Q1: How often do you copy-paste code during development (includes both intra- and inter-project)?

A1: Never: 3; Sometimes: 69

Q2: While fixing a bug in a code snippet, do you actively search for similar code snippets in the project?

A2: Yes: 53; No: 19

Q3: For developers who answered "yes" to Q2: How often do you search for similar code snippets while fixing bugs?

A3: Always: 21; Sometimes: 32

Q4: Would you like to have a clone detection tool as part of your development process?
A4: Yes: 65; No: 7

Q5: For developers who answered "yes" to Q4: Would you prefer a "Stand-alone/batch" or a "Plug-in/real-time" style clone detection tool for your use?
A5: Stand-alone/batch: 5; Plug-in/real-time: 60

Summarizing the survey responses, we find the following:

1. 95% of the participants copy-paste code during software development or maintenance.
2. 73% of the participants while fixing a bug in a code snippet, search for similar code snippets.
3. 90% of the participants would like to have a clone detection tool during software development.
4. 92% of the participants who want a clone detection support during development, would prefer that the tool be integrated as part of their development environment.

While this survey is not exhaustive, it clearly highlights software practitioners' positive preference for clone detection tool support integrated with the software development environment.

More details about the responses to the survey questions are available at [34].

6.3 SourcererCC-I's Architecture

This section provides an overview of SourcererCC-I's architecture. It has five modules as shown in Figure 6.2. We describe the function of each module below.

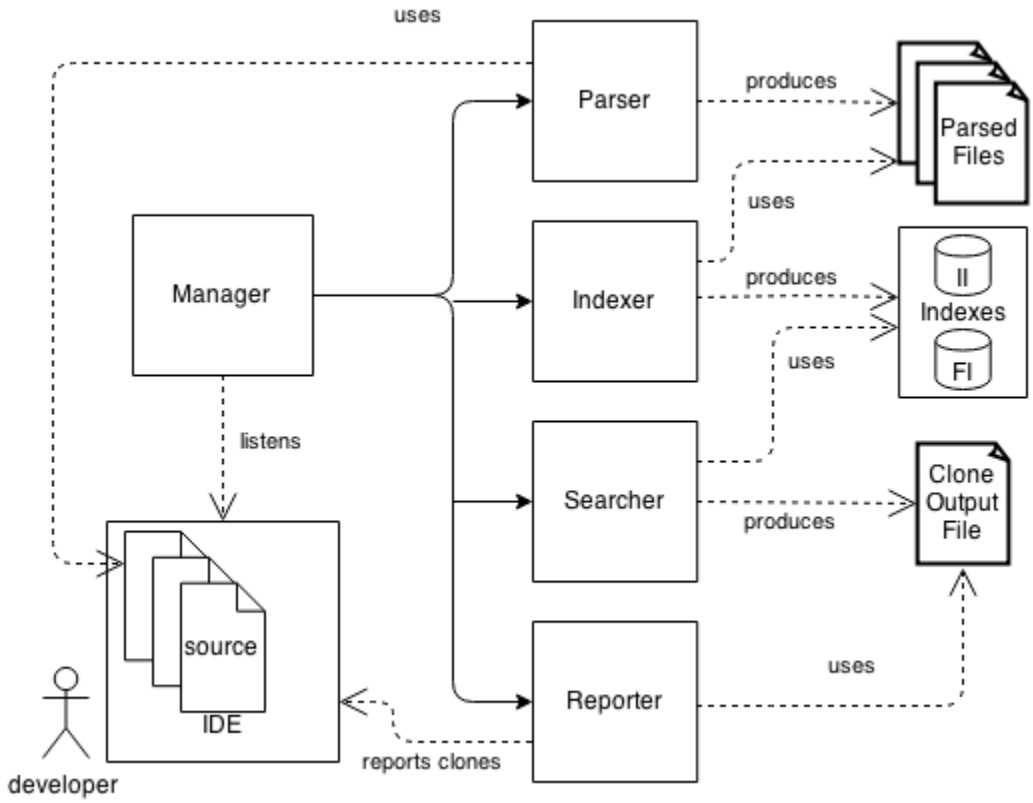

Figure 6.2: SourcererCC-I's Architecture

1. Manager: controls interaction among different modules

Manager module acts as a controller, delegating tasks like *create indexes*, *update indexes*, *search clones*, and *report clones* to other modules. It mediates the data flow among them and listens to the *change* and *selection* events generated by the editor. On detection of such events, it performs action such as *update indexes* or *search clones*.

2. Parser: parses projects and creates input for the indexer

Parser generates input files consisting of tokenized code blocks for the *Indexer* Module. On activation of the plug-in, *Manager* delegates the job of parsing the source files of all the projects in the active workspace to the *Parser* Module, which creates parsed files using Eclipse's JDT (Java Development Toolkit). The parsed files are now ready for the indexer to build index.

3. Indexer: creates inverted and forward indexes of code blocks

Indexer uses the parsed files to create a partial inverted index and a forward index for each of the open projects in the Eclipse's workspace. The partial inverted index is used to search the candidate clones whereas the forward index is used to verify if the candidates are clones or not. More details on the creation and working of these indexes can be found in Chapter 3.

4. Searcher: searches clones in the indexes

Manager detects the current method on which the developer is working on and then creates a query block for the method. *Searcher* uses this query block as an input and detect its clones using the indexes already built by the indexer. It produces clone output file containing meta information of clones including their fully qualified names, time to detect clones, and number of clones detected (clone group).

5. Reporter: reports detected clones

After the *Searcher* detects the clones of the current method, the *Manager* asks the *Reporter* module to perform two tasks (i) create a marker on the editor (where the line numbers are written). This marker signifies the presence of clones of the current method in the project; and (ii) display the list of clone methods using a tree-view in the console. A user can quickly navigate back-and-forth to the clone methods by clicking any item in this list of clone methods.

6.4 SourcererCC-I's Features

1. Non-obtrusive User Interface

SourcererCC-I has a minimalistic design in order to minimize the cognitive burden on the user. Hence, it's user interface is designed keeping non-obtrusiveness as a primary aspect of the design.

The tool uses colored markers to notify the clones detected. A marker is either *red*, *blue*, or *green* in color where *red* signifies there are more than 10 clones of the current method in the project; *yellow* signifies there are five to 10 clones; and *green* signifies the existence of less than five clones Figure 6.3 shows a red colored marker (annotated inside the top blue rectangular region in the Eclipse's editor) signifying that SourcererCC-I has found more than 10 clones of this method.

The output of most of the detection tools is impossible to read by a human at any large-scale. This is because of an extra step to link the output produce by the tools to the actual source of code fragments. To overcome this limitation, SourcererCC-I displays all the clones found for a method using their fully qualified name in the result view pane of Eclipse's console.

This not only facilitates easy navigation to cloned methods in the editor by simply using a mouse click, but also displays the clones in an hierarchical fashion using a tree structure. The detected clones are further grouped by the files and the projects in which they are present, thus making it easier for a developer to navigate.

2. Instant Clone Detection

SourcererCC-I builds an optimized index of all projects in the workspace once Eclipse is launched. Since the index creation phase is highly optimized using SourcererCC's filtering heuristics, it takes only few milliseconds even for projects having several thousand lines of code. Therefore, by the time a developer is ready to begin her development activity, index is already constructed. Now, as the developer starts typing, SourcererCC-I identifies the method in which she is operating and instantly detects all the clones of the method using the constructed index.

Due to SourcererCC-I's fast index creation process, even for a project consisting of several thousands of lines of code, clone detection process can begin as soon as the project is loaded in the workspace.

3. Incremental Index Creation

Software development is a very iterative activity, meaning, developers often change code that they have already written to either fix bugs or add new features. As a result, new methods are added, deleted or modified. Clone detection tools should account for these changes as clones should be detected on the latest version of the program and not any old version. For index based tools this means that the index needs to be recreated to account for the changes in the source code. Creating fresh indexes for the entire project with every change in the project can be very time consuming, and it can eventually slow down the

whole clone detection process. To overcome this issue, SourcererCC-I creates incremental indexes by keeping track of changed methods. As a result, the index is not re-created for the whole project whenever a method is changed, but it is re-created only for the changed method. This feature allows SourcererCC-I to instantly detect clones on the latest version of the project.

4. Inter- and Intra-project Clone Detection

A developer is often a part of multiple projects at the same time, or perhaps a part of a big project that consists of a family of many sub-projects. Under such scenarios, it is desirable for the tool to have the capability to search for similar code methods across the entire code base. To facilitate this requirement, SourcererCC-I has not only the ability to detect clones in a given project, but it can also seamlessly detect clones across projects or repositories.

5. Configurable Parameters

The granularity of clone detection usually depends on the task at hand and may vary according to different use cases. Moreover, sometimes, a developer might be interested in detecting clones that are more stricter in similarity (e.g., exact methods), whereas sometimes she might be interested in detecting similar methods that have experienced significant editing. As a result, SourcererCC-I allows developers to specify the granularity and similarity threshold parameters in its configuration file.

SourcererCC-I is demonstrated to have optimal recall and precision at 0.7 similarity threshold for methods and at 0.8 for files. Hence, the default configuration is to detect method level clones at 0.7 similarity threshold.

6. Scalable to Large projects and Repositories

SourcererCC-I is developed using SourcererCC's API. SourcererCC is demonstrated to have successfully detected *all* the clones in a repository consisting of 25,000 projects containing 3 million files and more than 250 MLOC on a standard workstation with reasonable hardware specification. Moreover, we envision that in most cases, a developer is only interested in detecting clones of the method that she is focusing on at a given time and not all the clones in a project. As a result, the usage scenario of SourcererCC-I is not as extreme as SourcererCC's further enhancing SourcererCC-I's scalability.

7. Quick and Easy Setup

SourcererCC-I is developed as an Eclipse plug-in and hence can be installed easily by following these 3 steps: (i) Open Eclipse and click *Install New Software* in *Help* menu; (ii) Click *Add*. Then, in the *Add Repository* window, enter the name SourcererCC-I, and in the location field enter the url `http://mondego.ics.uci.edu/projects/clonedetection/tool/latest/`; and finally, (iii) follow the steps given in Eclipse's wizard to install the plug-in.

6.5 Related Tools

We have extensively covered the work related to clone detection in the previous chapters. Here we focus on the tools which are closely related to SourcererCC-I.

Tools integrated with the development environment. Patricia Jablonski's proposed CnP, a tool to detect copy-and-paste clones in the IDE [48]. CnP establishes links between the original and the pasted clones and uses their content information later on for error detection and other purposes. Unlike SourcererCC-I, CnP does not use a clone detector to

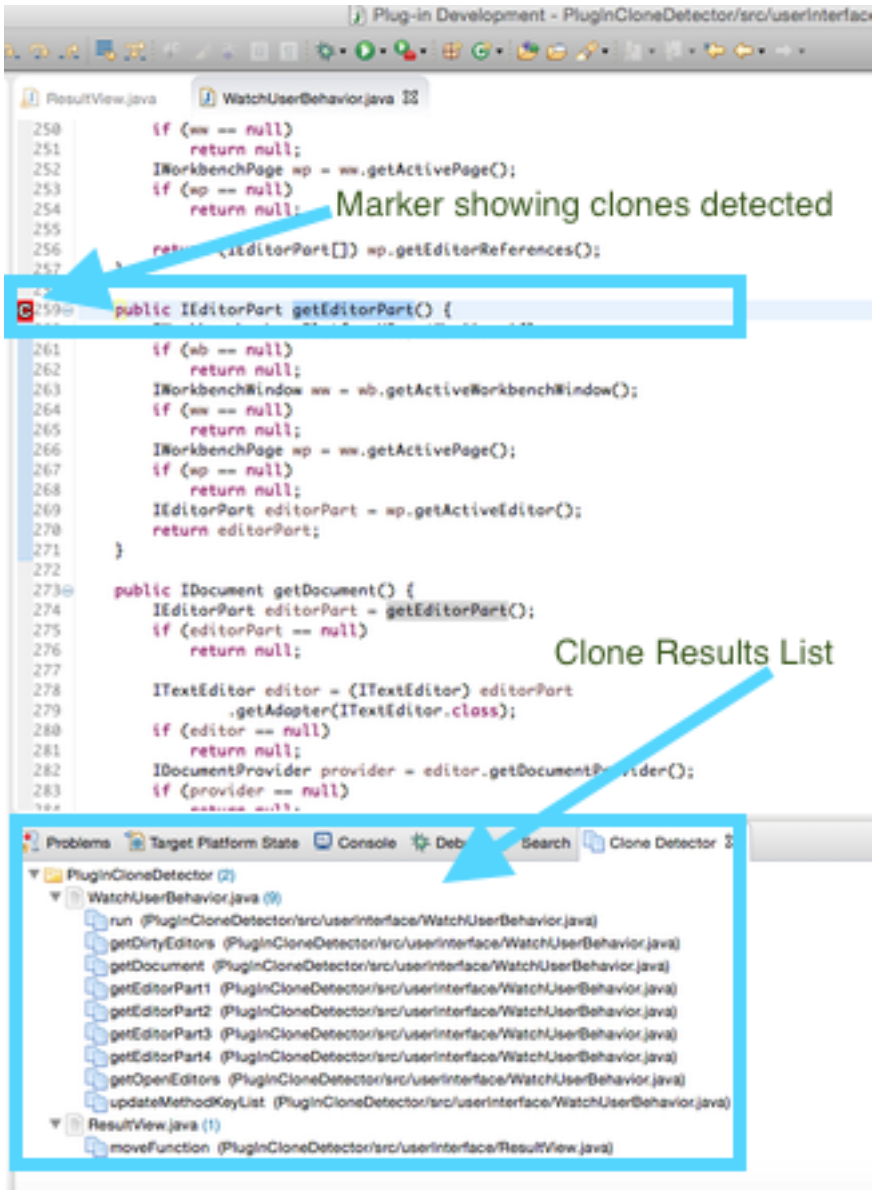

Figure 6.3: Eclipse's screenshot showing clones detected using SourcererCC-I

detect clones and hence is not capable of finding legacy clones. Moreover, CnP cannot find accidental clones.

CP-Miner is a commercial tool that detects copy-paste errors in the context of traditional clone detection [82]. It uses a token-based approach with data mining and a gap constraint. The tool is tailored specifically for detecting bugs and is not useful for other purposes like code refactoring or re-use.

CloneTracker, an Eclipse plug-in is a tool that keeps track of the evolution of clones [35]. For it's input, it relies on the output of a clone detector tool, that needs to be run in advance to generate all the clones in a system. CloneTracker, unlike SourcererCC-I, does not detect clones in real time.

Stand-alone Tools. There are few well packaged stand-alone batch tools, namely Dup [7], iClones [41], CCFinder [62], and NiCad [105] to detect clones, but these tools are not designed to be integrated with the development process. NiCad, however, claims that it can be integrated with an IDE but being a batch processing tool it lacks the capability to detect clones instantly. Baxter et al. created an abstract syntax tree based clone detector that works by finding identical subtrees [12]. The tool is now commercialized (CloneDR) and has been demonstrated to have high precision at the cost of low recall.

6.6 Tool Artifacts

The tool artifacts are made available for public use.

Link to install the Eclipse plug-in: `http://mondego.ics.uci.edu/projects/clonedetection/tool/latest`

Link to the demo: `https://youtu.be/l7F_9Qp-ks4`

6.7 Chapter Summary

While there exists a large number of clone detection tools, very few are suitable for developers' needs because: (i) they come as stand-alone tools and thus cannot support clone-aware development; (ii) they are ineffective in detecting near-miss (Type 3) clones; and (iii) they are unable to perform instant clone detection.

This chapter presents SourcererCC-I, an easy-to-use near-miss clone detector plug-in for Eclipse IDE that can detect clones on the fly. The tool pro-actively finds code snippets a developer is focusing on and non-intrusively report its clones in real time. Since the tool is built on top of SourcererCC, it has the ability to efficiently scale to large code bases and or software repositories.

SourcererCC-I is an ongoing attempt to bring the academic clone research closer to the real-world developers. In the future, we plan to extend SourcererCC-I into a clone management workbench by incorporating features such as clone tracking and semi-automated clone refactoring support.

Chapter 7

Empirical Applications of SourcererCC

Part of the material in this chapter is included with the permission of the IEEE and based on our work in:

- Sajnani, H.; Saini, V.; Lopes, C., "A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code" Source Code Analysis and Manipulation (SCAM), 2014 14th IEEE Working Conference on, pp. 21-30, 28-29 Sept. 2014 doi: 10.1109/SCAM.2014.12

Large-scale empirical studies on code cloning depend on the quality and scalability of clone detection tools. One of the primary motivations behind the development of SourcererCC is to provide the infrastructure to enable high quality empirical clone research.

In order to demonstrate the effectiveness of SourcererCC on this front, this chapter presents two empirical studies conducted using SourcererCC. The first study, "A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code" [110], was led by the author whereas

the second study, "A Comparative Study of Software Quality Metrics in Java Cloned and Non-Cloned Code" was led by Vaibhav Saini who is a collaborator on the SourcererCC project.

In this chapter, I will describe the first study in detail, and briefly given an account of the key aspects of the second study and how SourcererCC was used in the study.

7.1 Introduction

Over the last couple of decades, software development practices have changed drastically. Pervasive high-speed Internet, full fledged IDEs, and a whole new generation of hyper-connected young programmers weaned on the web have established new programming practices based on massive collaboration. These days, it is easier than ever to find and use a well-tested piece of code written by someone else that does exactly what we want. Gluing these pieces together and putting them in the right context is still a necessary and important skill. But, for better or worse, copy-and-paste is no longer a pejorative term, but a factual observation about how a part of modern coding gets done today. Reusing code fragments via copy-and-paste, with or without modifications or adaptations, also known as code cloning, has become a common behavior of software engineers [107].

Although pervasive, code cloning has traditionally been criticized by researchers and leading practitioners alike. Parnas [97] notes that *"if you use copy and paste while you're coding, you're probably committing a design error."* Indeed, if instead of copying code, we move it into its own method, future modifications will be easier because we will need to modify the code in only one location. The code will be more reliable because we will have only one place to ensure that the code is correct. Consequently, code cloning is often presented as a negative design characteristic in software systems. Considered as a bad *"code smell"* [38],

a considerable amount of research in cloning is concentrated on detecting clones in existing source code [11, 5, 36, 56, 60, 69, 76, 42, 73], removing them [70, 72] and refactoring them [112].

Under many circumstances, code cloning can, indeed, be harmful. But it also has advantages like rapid development, reuse of tested code, and separation of concerns. The pervasive practice of code cloning has more recently attracted researchers to conduct empirical studies to find evidence about the effects (good or bad) of code cloning. Such attempts have questioned our conventional wisdom about the harmful nature of clones. For example, Kasper and Godfrey presented evidence that clones may be intentional and that they improve developer productivity [63]. Kim et al. [68] found that most of the clones are short lived – i.e. starter code that quickly becomes something else – and hence investment made in refactoring them may not be worth the effort. Toomim et al. [121] showed that managing clones via linked editing to edit multiple cloned regions without much programmer intervention can be an efficient way of dealing with clones. Rahman et al. [98] conducted a study on four subject systems to assess the impact of clones on defect occurrence of software products and did not find any evidence that cloning is harmful.

These findings present a different perspective on code cloning that has implications for the future research in this area. Although important, most of the research in this direction, so far, is either qualitative or performed on very few subject systems. An excellent survey [104] on code clone research mentions that *"there is little information available concerning the impacts of code clones on software quality"*, expressing the need to conduct more empirical studies examining the impact of code cloning on various factors related to code and other artifacts. Koschke [71] lists several important open issues in this field, one of which being *"What is the relation of clones to quality attributes?"*

To that end, this chapter presents two empirical studies conducted using SourcererCC to explore the relationship between code clones and various quality attributes.

Study 1. *A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code*

We conduct an empirical study of 31 open source Java projects (1.7 MSLOC) to explore the relationship between code clones and a set of bug patterns reported by FindBugs. We found that: (i) the defect density in cloned code is 3.7 times less than that of the rest of the code; (ii) 66% of the bug patterns associated with code clones are related to issues in coding style and practice, the two least problematic of the Find Bugs' categories, while that number is 49% for non-cloned code; and (iii) 75% of the bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones. These results show that, when using FindBugs to detect bug patterns, there is a positive differentiation of cloned code with respect to the rest of the code: the cloned code has considerably less, and less problematic, bug patterns.

Study 2. *A Comparative Study of Software Quality Metrics in Java Cloned and Non-Cloned code*

This study was primarily conducted by Vaibhav Saini, one of the collaborators of SourcererCC project as part of his advancement to candidacy at University of California, Irvine.

The study was conducted on 4,421 open source Java systems to explore the relationship between code clones and a set of 27 software quality metrics spanning across three categories: complexity, modularity, and documentation (code comments). It was found that for most of the systems, complexity and modularity of the cloned methods is better than the non-cloned methods. Non-cloned methods, however, are found to have higher number of code comments.

7.2 Study 1. A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code

This study examines the relationship between clones and various bug patterns reported by FindBugs [37]. A bug pattern is a code idiom that is often a programming error and hence an indicator of code quality. These bug patterns arise for a variety of reasons including difficult language features, misunderstood API methods, misunderstood invariants when code is modified during maintenance, and variety of mistakes including typos, use of the wrong boolean operator, etc. Hence they capture various kinds of issues impacting the quality of code. Moreover, these bug patterns are organized into high level categories like *Bad Practice*, *Correctness*, *Performance*, etc. We use this categorization to establish the relationship between clones and specific bug categories. We posit that such analysis will help us to investigate the associations between code quality and cloning, which in turn, will be useful to inform research in this field.

7.2.1 Research Questions

We seek answers to the following research questions.

Research Question 1: *Is defect density of cloned code greater than that of non-cloned code?*

For the purpose of this study, we consider a piece of code to be cloned if there exists a similar piece of code in the same system (intra-system clones only).

We use bug patterns reported by FindBugs to calculate the defect density, where defect density is defined as the number of bug patterns reported by FindBugs per 1,000 lines of code. FindBugs pro-actively reports likely defect locations in code by using range of

approaches from simple code pattern-matching techniques to rigorous static analyses that process carefully designed semantic abstractions of code. Thus FindBugs *modus operandi* is to automatically prove certain properties of a program, i.e. certifying the program free of a certain class of bugs. Since, large scale studies like ours are heuristic in nature, FindBugs gives us the automation we need for this much larger dataset. We discuss more about the rationale of using FindBugs' bug patterns in the context of this study in Section 7.2.2.3.

This research question is the main topic of our study. Clearly, if we were to find a much higher defect density in cloned code than in the rest of the code, there would be irrefutably strong arguments against the practice of cloning.

Research Question 2: *Are there specific bug categories which are seen more often in the cloned code?*

Each bug pattern reported by FindBugs is associated with a category. Since each category poses a different threat level and captures a different class of bugs, we try to examine the relationship between these categories and code clones. Such analysis is useful for risk assessment and employing targeted measures to mitigate the risks.

If we were to find that cloned code has higher rate of *Correctness, Performance* or *Security* bug patterns than the rest of the code, that would also be a strong argument against the practice of cloning.

Research Question 3: *How often do bug patterns propagate through cloning?*

The primary goal of this research question is to explore if code cloning leads to an increase in the total number of bugs in the system. If this is the case, each code block with a bug, when cloned, adds one more bug in the code-base; eventually degrading the overall code quality. One implication of this finding could be building tools and techniques to help developers pro-actively improve code snippets (e.g., fix bugs) before copying it to some other location.

Outline: The remainder of this section describes the study design (Section 7.2.2), results and examination of statistical differences (Section 7.2.3), and the threats to validity (Section 7.4). Finally, we summarize our findings (Section 7.2.4).

7.2.2 Study Design

At a high level, the overall approach to analyze the bug patterns in the code clones is a two step process. In the first step, we use a clone detection tool to identify all the clones present in each subject system. We set the granularity of the tool to detect method level clones. Thus, we have a list of methods for which clones exist in the system. Similarly, we also have a list of methods for which no clones exist in the system. A set of methods which are clones of each other is called a code clone group. Each member of a group is a clone sibling.

In the second step, we find all the bug patterns present in the subject system using FindBugs. After detecting all the bug patterns in the system, we create a map to associate methods with the bug patterns found in them. Since, each bug pattern belongs to a high level bug category, we also create a map to associate methods with bug categories.

Combining the result of the previous two steps, we have, for each method in the subject system:

1. All the other methods which are clones of this method (clone siblings); and
2. All the bug patterns and bug categories found in this method

We use this information and statistical analysis to seek answers to the research questions posed above. In the remainder of this section, we describe each aspect of our study design in detail.

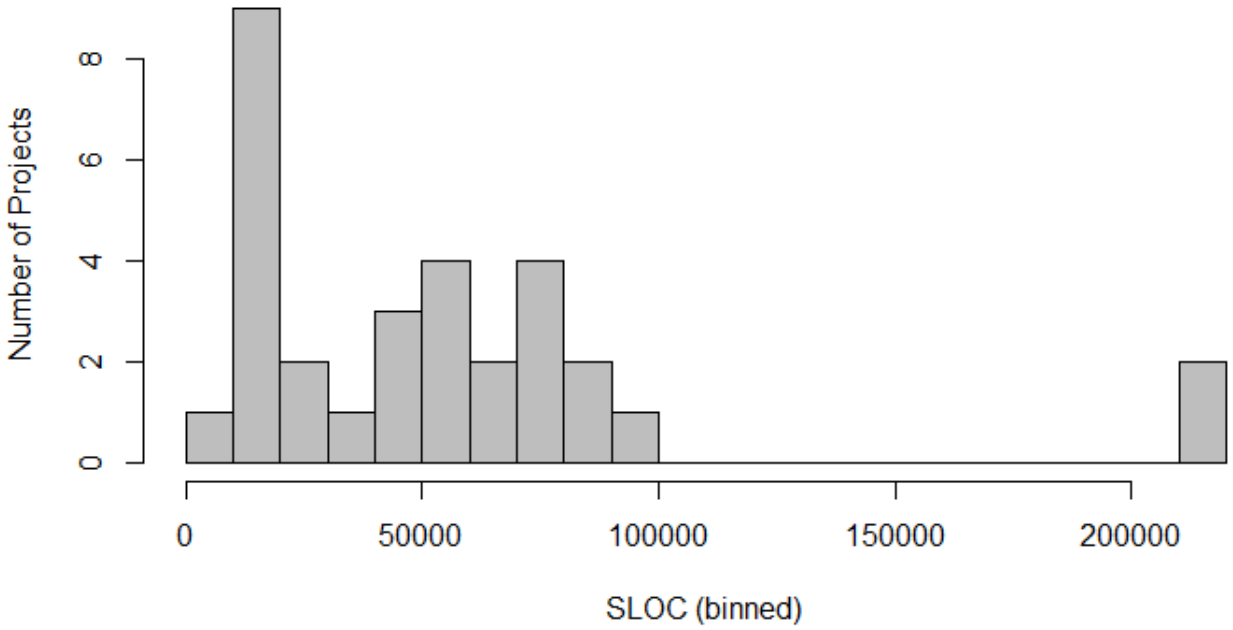

Figure 7.1: Size distribution of projects. Size is defined as non-commented lines of code

7.2.2.1 Subject Systems

We chose 31 open source Apache Java projects as subject systems for this study. These projects are of varied size and span across various domains including search and database systems, server systems, distributed systems, machine learning and natural language processing libraries, and network systems. Most of these subject systems are highly popular in their respective domain. Such subjects systems exhibiting variety in size and domain help counter a potential bias of our study towards any specific kind of software system.

Figure 7.5 describes the size distribution of the subject systems. The X-axis represents the binned uncommented lines of code (SLOC). The Y-axis represents the number of projects. The average project size is 54,382 SLOC. The name and exact size of each project is listed in Column 1 & 2 of Table 7.1 respectively.

7.2.2.2 Clone Detection

We use SourcererCC with 0.7 similarity threshold (θ) to detect method level clones because in our evaluation (Chapter 4) it gave maximum recall and precision.

Precision. In order to evaluate the quality of clones detected, we chose a statistically significant random sample of clone siblings for manual inspection. The author examined 140 clone siblings and classified them as either true positives or false positives. We discovered only four false positives. Hence, using a t-test, we can conclude with 95% confidence that the false positive rate of the clone detection tool is only $2.3 \pm 5\%$.

Moreover, on further examining the false positives, we found they consists of empty methods which are *stubs* without any functionality. To further safely avoid such cases, we included only those methods which had at least 20 tokens in the body. This step eliminated all the false positives in the previous sample, further increasing precision without impacting recall.

Columns 5 & 6 in Table 7.1 respectively list the *total methods* and *cloned methods* found in the subject systems and Columns 4 & 7 show the percentage of total SLOC and percentage of total methods clones in each subject system.

7.2.2.3 Why use Bug Patterns?

Measuring external quality can be exceedingly challenging, as it requires identifying defects and inaccuracies in software. Such defect identification is usually done by either using bug reports or by using static analysis tools like FindBugs.

While both these techniques of defect identification have their merits [99], in the context of this study, there are several reasons why we choose bug patterns detected by FindBugs to measure the external code quality.

First, defect identification relies heavily on links between bug databases and program code repositories. This linkage is typically based on bug-fixes identified in developer-entered commit logs. Unfortunately, developers do not always report which commits perform bug-fixes. Moreover, developers sometimes fix bugs that are only reported in some other projects' bug tracker, rather than in their own; and vice-versa. Thus only a fraction of bug fixes are actually labeled in source code version histories. The question naturally arises, are the bug fixes recorded in these historical datasets a fair representation of the full population of bug fixes? Bird et al. [15] investigated historical data from several software projects, and found strong evidence of systematic bias. They found that bias is a critical problem that threatens both the effectiveness of processes that rely on biased datasets to build prediction models and the generalization of hypotheses tested on biased data.

Second, apart from completeness of data, there are also serious concerns regarding the quality of bug-fix data obtained in such manner. Herzig et al. [46] found 33.8% of all issue reports to be misclassified, that is, rather than referring to a bug fix, they resulted in a new feature, or an internal refactoring. This misclassification introduces bias in bug data, confusing bugs and features. They manually examined 7,000 issues, and found that on average, 39% of files marked as defective, actually never had a bug.

Third, and more importantly, the number of reported defects is heavily correlated to the popularity of a project [45]. The more users a project has, the more people there are discovering and reporting defects. A project, only used by a handful of people, may be significantly lower in quality than a project used by tens of thousands of people, but popular projects will almost certainly have more bugs reported. In contrast, the number of bug patterns in a project and project popularity are not found to be correlated [111].

Fourth, similar to the third issue, affects the number of reported bugs based on the life of the project. A very old project may have hundreds of reported bugs, however, a project that has just started may have fewer bugs, if any. Thus practically making it impossible to

compare them.

FindBugs [37] uses heuristics and static analysis to identify common bug patterns in software that may result in externally visible defects. Any instances of these patterns are then reported to developers as potential bugs. While such systems make no guarantees as to correctness or completeness, studies have shown that they regularly identify important defects in software [4, 49]. Also, since large-scale studies like ours are heuristic in nature, FindBugs gives us the automation we need for this much larger dataset. Hence, using FindBugs to look at the exercise of code cloning will provide us with one more perspective to understand the practice of code cloning better.

Credibility of FindBugs. FindBugs is a heuristic tool; as such, it suffers from both false positives and false negatives. False positives (i.e. reported bugs that aren't really bugs) are particularly problematic. A previous study has shown that FindBugs results in slightly less than 50% false positives [49], which is a high rate. Nevertheless, the rate of false positives in FindBugs affects the cloned code and the non-cloned code in about the same way. Since our goal is a comparison of these two sets of code, our use of FindBugs is fair.

Second, FindBugs bug patterns are used to assess open source software on a regular basis. For example, in order to regularly perform scans of open source software, the U.S. Department of Homeland Security uses Coverity [29], a commercial high-end bug finding product, which includes several bug patterns from FindBugs.

Third, in 2009, Google held a global fixit for FindBugs tool that had interesting results[1]. The focus of the fixit was to get feedback on the 4,000 highest confidence issues found by FindBugs at Google, and let Google engineers decide which issues, if any, needed fixing. More than 700 engineers ran FindBugs from dozens of offices. More than 250 of them entered more than 8,000 reviews of the issues. A review is a classification of an issue as

[1]Information published on FindBugs website: http://FindBugs.sourceforge.net/

must-fix, should-fix, mostly-harmless, not-a-bug, and several other categories. More than 75% of the reviews classified issues as must-fix, should-fix or I-will-fix. Many of these issues received more than 10 reviews each. It is reported that engineers submitted changes that resolved more than 1,100 issues by 2010. As reported, the work continues on addressing the issues raised by the fixit, and on supporting the integration of FindBugs into the software development process at Google.

These observations instill confidence that the bug patterns reported by FindBugs, even if not measuring external quality directly and deterministically, are positively correlated with it.

Bug Pattern Categories. FindBugs classifies each bug pattern into a specific category which determines the type of the bug pattern. These categories include *Correctness*, *Multi-threaded Correctness*, *Performance*, *Security*, *Malicious Code*, *Style*, and *Bad Practice*. A short description of each bug pattern along with its category can be found at *http://FindBugs.sourceforge.net/bugDescriptions.html*.

Limiting FindBugs' False Positives . One of the problems with the code quality tools like FindBugs is that they tend to overwhelm developers with problems that may not really be problems i.e., false positives. Although FindBugs is reported to have less than 50% false positives [49], we believe that large numbers of false positives can skew the results of the analysis. Hence, in order to mitigate this issue, we take the following steps:

We create two meta categories of bug categories in FindBugs. The first category, which we call Primary category, includes bug categories like *Correctness*, *Multi-threaded Correctness*, *Performance*, and *Security*. FindBugs reports that the bug patterns reported in these categories are precise and actionable.

The second meta category, which we call Secondary category, includes categories such as *Bad Practice* and *Style*. Since FindBugs accepts more false positives in these categories, there

are cases when one might decide that a bug pattern is not relevant for one's code base. For example, one never uses Serialization for persistent storage, so one never cares about the fact that one didn't define a serializationUID. But FindBugs would report this bug pattern anyway. Moreover, even for the bug patterns which are relevant to one's code base, perhaps only a minority will reflect problems serious enough to change the code. So in most cases, Secondary category does not pose a threat other than affecting the readability of code. Thus we conduct our analysis separately for Primary and Secondary category whenever relevant to avoid potential threats due to the nature of bug patterns the two categories detect.

Also, FindBugs assigns a severity level (HIGH, MEDIUM, or LOW) to each bug pattern based on its threat. To further mitigate the risk of false positives, we configured FindBugs to exclude LOW severity bug patterns. Examples of such low-severity bug patterns are *field names not starting with lower-case*, and *method that fails to close stream on exception*. The complete list of such bug patterns is available at FindBugs's website.

7.2.3 Study Results

In this section, we present the results of our analyses of code clones and their relationship with a set of bug patterns reported by FindBugs. Our findings are mostly consistent across all the subject systems and we describe the places where they are different. Each sub-section below specifically addresses the research questions posed in Section 7.1.

7.2.3.1 RQ1: Is defect density of cloned code greater than that of non-cloned code?

To answer this question, we compare the defect density of cloned code with non-cloned code across all the projects. Column 4 (Defect Density) in Table 7.1 shows the defect density

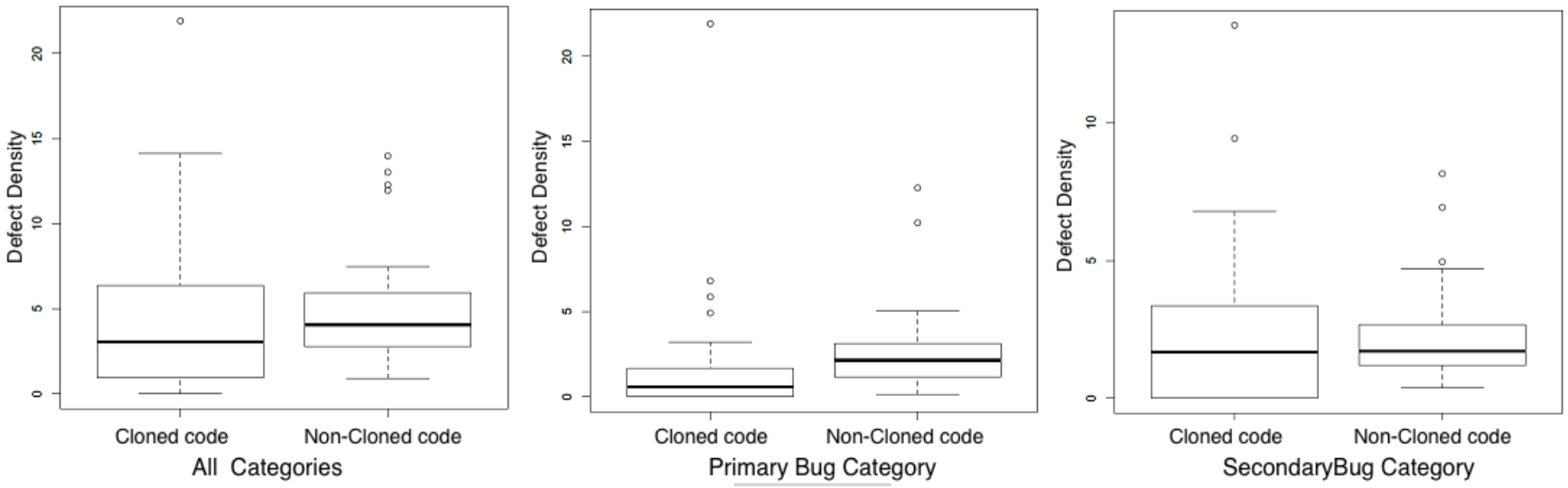

Figure 7.2: Box plot showing defect density of cloned-code and non-cloned code. Note that defect density of cloned-code is less than that of non-cloned code for all the categories (left). For primary category, the defect density of cloned-code is 3.7 times less than non-cloned code (center). For secondary category the difference is zero (right), implying that most of the bugs in cloned code are of secondary category which consists of least problematic categories in FindBugs.

of cloned and non-cloned code for each project respectively. We found that in 26 out of 31 projects, the defect density of cloned code is lower than that of non-cloned code.

As discussed, Primary and Secondary category differ in the severity and the type of bug patterns they detect. Secondary category consists of bug patterns related to *Style* and *Bad Practice*. In order to avoid the threat of such non-crucial bug patterns impacting RQ1, we only consider Primary category for our analysis to answer RQ1. However, for the sake of completeness, we also describe the data for all the categories as well as separately for Primary and Secondary category.

Figure 7.2 shows three box plots comparing median defect density (Y axis) in cloned code and non-clone code across three groups consisting of "All", "Primary", and "Secondary" bug category (l-r). As shown in boxplot (left), considering all the categories, the difference in the median defect densities of non-cloned (4.06) and cloned code (3.06) is 1. The difference decreases to almost 0 (0.02) when we consider only secondary bug-category (right boxplot). This implies that the observed difference in the median defect densities when considering all the categories is mainly because of the primary category - which is our category of interest because it consists of severe and more problematic bug patterns. For the primary category, the median defect density of the cloned code and non-cloned code is 0.58 and 2.14 respectively (center boxplot), making the defect density of cloned code 3.7 times less than that of non-cloned code.

Many observable program features correlate strongly with code size. This knowledge has been used pervasively in quantitative studies of software through practices such as normalization on size metrics. We wanted to explore if the number of bug patterns in a method also follow a similar trend. Since the total size of cloned code is much smaller than that of non-cloned code, we considered the possibility that this size difference is serving as a confounding factor. Moreover, we found that the average method size of clones is 17 LOC and that of non-clones is 7 LOC. As such, the following two factors can impact the result of our analysis: (i) the

relative difference in the average method size of cloned and non-cloned code for each project; and (ii) the relative difference in the total number of cloned and non-cloned methods for each project.

In order to understand the impact of method size on our analysis, we first compute the Pearson correlation coefficient between method size (LOC of a method) and the number of bug patterns found in a method. The goal is to determine how strongly the number of bug patterns in a method correlate with the method size. Clearly, if the number of bug patterns show a strong positive correlation with the method size, then the code group (clone/non-clone) with bigger methods i.e., cloned code, is likely to have more more bugs.

The correlation between bug patterns and method size for all the methods in the corpus is 0.151; 0.108 for only cloned methods, and 0.147 for only non-cloned methods . Figure 7.3 shows the scatter plots between bug patterns and method size for all the methods, only cloned methods, and only non-cloned methods (l-r) transformed on a log-scale. Manual inspection of the scatter-plots confirmed that the correlations fell near zero and represented pairings without an identifiable relationship. We found that the distribution of bug patterns in the project is right skewed, meaning that majority of the methods contain very few bug patterns. This result is not unexpected, and confirms the intuition that some methods have many bug patterns while most of the methods contain hardly any bug pattern at all. This heavy right skew also means that scatter plots between bug patterns and method size will have the majority of points clustered at low values if a log transformation is not done.

Column 5 (Correlation) in Table 7.1 shows the correlation coefficient for each project individually. As shown, most of the projects show very low value of correlation coefficient.

The fact that method size is very weakly correlated with the number of bug patterns found in that method, is an indication that method size may not be serving as a strong confounding factor. While the above experiment ensures that the difference in the average method size

of clones and non-clones does not impact the result of RQ1, the large difference between the total number of cloned and non-cloned methods might still pose a threat to our analysis. In order to address this, we performed the following two experiments.

In the first experiment, for each project, we randomly pick, from the pool of non-cloned methods, only as many methods as the total cloned methods in the project. The goal is to have equal number of methods in both the groups and then compare their defect densities. We found that the average defect density of cloned code to be 4 times less (0.58 vs. 2.32) than that of non-cloned code.

In the above experiment, although, both the code groups have the same number of methods, because of the difference in the average size of method in each group (17 LOC vs. 7 LOC), the average LOC in each group may be different. Hence, in order to further ensure the validity of our analysis, we modify the above experiment to randomly pick non-cloned methods, one by one, such that the total LOC of non-cloned code group is same as that of cloned code group. Note that in this case, the total number of methods in each group may be different, but, each group will have same LOC. We found that, even under this setting, the defect density of cloned code is 3.1 times less (0.58 vs. 1.82) than that of non-cloned code.

Columns 6 (Defect Density (size control)) in Table 7.1 shows the defect density of non-cloned code for each project separately for each experiment. Note that the results are averaged over 100 runs to ensure sufficient randomization while picking the non-cloned methods. Below we show statistical significance of the difference.

Statistical significance of the difference. We use statistical paired tests to study the significance of the difference between median defect densities of the two groups. We formulate the hypotheses as follows:

$$H_0 : \mu(r_{NC} - r_C) <= 0, H_A : \mu(r_{NC} - r_C) > 0$$

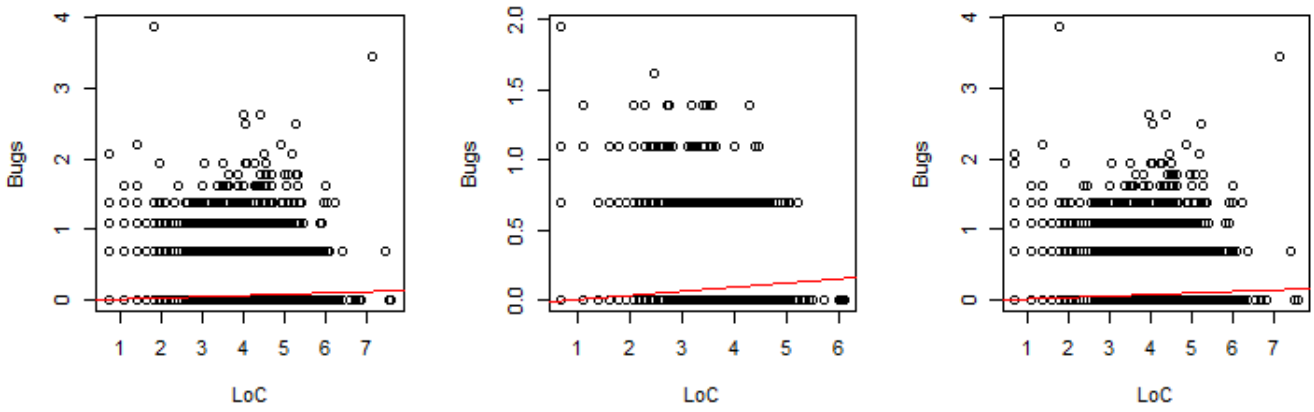

Figure 7.3: Scatter plots of method size and bug patterns for only cloned, only non-cloned, and all the methods (left-right) reveal no identifiable relationship.

where $\mu(r_{NC} - r_C)$ is the population median of the difference in defect density between the non-cloned code (NC) and the cloned code (C) for each project.

H_0: The median defect density of non-cloned code is less than or equal to that of the cloned code,

H_A: The median defect density of non-cloned is greater than that of cloned code.

For our analysis, we conduct Wilcoxon signed rank test because it is a non-parametric paired statistical test and does not make any assumptions about the normality of the data [85]. The computed p-value is $0.0031(<< 0.05)$. We note that the median defect density in the clone code is 0.58 and that of the non-cloned code is 2.14. Large scale studies are often guaranteed to give small p-values, therefore we are also reporting the summary statistics (medians), so that the reader can judge the significance of the differences.

The defect density computed using FindBugs in cloned code is 3.7 times less than that of the rest of the code, implying that there is a positive differentiation of cloned code with respect to the rest of the code

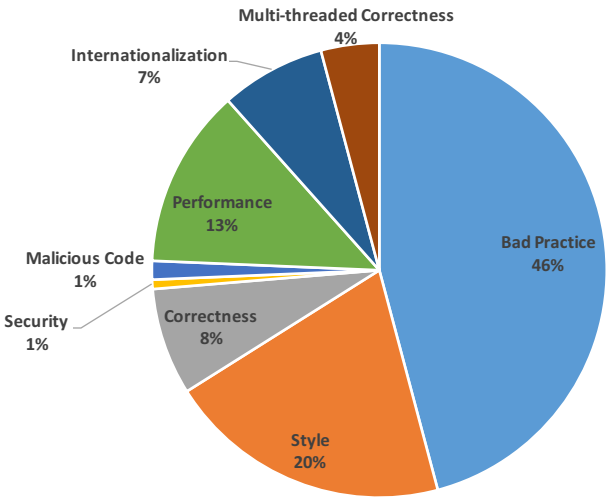

Figure 7.4: Bug patterns in cloned code classified into various categories

7.2.3.2 RQ2: Are there specific bug categories which are seen more often in the cloned code?

To answer this question, we examine the relationship between bug categories and cloned code. Since each category poses a different threat level, such analysis will be useful for risk assessment and employing targeted measures to mitigate the risks.

Figure 7.4 shows how bug patterns found in the cloned code are distributed across various categories. While there are bug patterns in all the categories, as seen from a total of 399 bug patterns, 66% (26 + 40) of bug patterns found in the cloned code belong to *Style* and *Bad Practice* categories (Secondary category). Although not shown in the chart, this number is as low as 49% (2,065 out of 4,206) for non-cloned code. It is worth noting that these bug patterns are mostly violations of recommended and essential coding practice and are susceptible to more false positives. This implies that not only cloned code have fewer bug patterns compared to non-cloned code, most of the bug patterns belong to the two least problematic category of FindBugs.

Statistical significance of the difference. In order to statistically evaluate the dominance

of Secondary category in the cloned code, we formulate the following hypotheses:

$$H_0 : \mu(r_P - r_S) >= 0, H_A : \mu(r_P - r_S) < 0$$

where $\mu(r_P - r_S)$ is the population median of the difference in bug pattern count between Primary category (P) and Secondary category (S) in cloned code.

H_0: The median bug pattern count of Secondary category is greater than or equal to Primary category.

H_A: The median bug pattern count of Secondary category is less than that of Primary category. the p-value obtained using Wilcoxon signed rank test is 0.0415. While the median number of bug patterns found in both the primary and the secondary category is equal to 2, the mean number of bugs found in the primary and secondary category are 6.67 and 12.87 respectively.

66% of the bug patterns associated with code clones are related to issues in coding style and practice, the two least problematic of the FindBugs' categories, while that number is 49% for non-cloned code

7.2.3.3 RQ3: How often do bug-patterns propagate through cloning?

In order to investigate how often bug patterns are duplicated when the code is cloned, we manually examined 339 code clone groups which had at least one bug pattern present in them. Note that a code clone group is a set of methods which are clones of each other. For each code clone group, we look at all the bug patterns present in the group one at a time, and classify the group into one of the following two categories.

Category 1 - If the bug pattern present in the code clone is also present in at least one other

Project	SLOC		#Method		Defect Density		Correlation	Defect Density (Non-clones) (size control)		Bugs	
	Total	Clones	Total	Clones	Non-clones	Clones		Equal # Methods	Equal LOC	duplicated	non-duplicated
ant	86,438	3,642	7,883	208	2.66	1.65	0.195	2.46	2.61	4	0
berkeleyparser	57,905	7,842	3,364	357	4.35	0.89	0.168	4.08	4.23	9	10
cglib	13,668	354	1,916	28	2.69	0	0.123	3.74	1.65	0	0
cloud9	56,766	21,066	4,390	1,047	11.96	6.5	0.149	5.02	5.07	–	–
cocoon	10,387	211	773	25	2.08	4.92	0.197	3.87	1.82	1	0
commons-io	8,673	440	674	50	1.05	0	0.137	2.97	3.56	1	1
dom4j	17,854	997	2,206	72	3.74	0	0.096	0.82	0.8	1	0
hadoop-hdfs	70,411	5,016	5,401	362	2.48	2.03	0.051	1.05	1	0	1
hadoop-mapred	64,023	3,452	5,577	230	3.05	1.28	0.205	2.4	2.25	2	1
hibernate	72,409	3,117	8,973	243	4.18	0	0.017	2.79	2.97	6	1
httpclient	18,022	1,206	1,324	102	3.06	6.8	0.101	1.6	1.81	2	0
jfreechart	93,460	15,470	7,034	871	1.01	0.2	0.214	0.89	0.72	89	31
jython	211,905	7,411	19,593	620	1.47	1.66	0	0.1	0.22	1	2
log4j	16,111	609	1,687	57	0.86	0.58	0.142	2.32	2.24	2	1
lucene	15,670	1,438	1,315	77	0.14	0.13	0.152	1.66	1.06	4	2
mahout-core	53,366	3,800	3,943	249	1.61	0	0.003	1.54	1.09	0	0
mason	35,931	3,800	2,674	261	1.48	0	0.278	1.37	1.36	8	5
nutch	12,243	519	861	40	4.24	0	0.299	4.49	4.34	1	0
pdfbox	13,936	1,653	894	31	12.27	0	0.256	14.81	9.74	0	0
pig	84,770	3,059	5,458	180	3.03	0.33	0.157	2.97	2.27	1	7
pmd	60,060	6,864	4,634	254	0.79	0.58	0.33	0.67	0.7	76	2
poi	47,804	5,093	4,899	343	4.87	2.16	0.117	4.8	4.18	6	9
postgresql	23,514	1,916	1,836	160	10.22	21.91	0.415	8.64	8.07	20	2
rhino	54,722	2,412	2,958	135	0.87	0	0.209	0.5	2.4	0	0
stanford-nlp	210,233	4,430	8,692	295	5.72	9.26	0.192	3.18	3.45	–	–
struts	24,799	1,275	2,300	82	1.61	0.78	0.127	1.87	1.24	0	1
substance	47,361	5,014	3,026	213	1.32	0.4	0.233	1.13	0.84	8	1
synapse-core	41,612	2,370	2,803	138	0.78	0	0.18	0.88	1	1	2
tomcat-catalina	73,673	4,030	4,757	232	2.32	0.99	0.171	2.26	2.76	5	2
uima-core	11,942	687	947	59	2.14	0	0.007	2.53	0.8	0	0
xerces	76,185	7,340	5,131	345	0.91	0.68	0.205	1	0.87	6	4

Table 7.1: Results. Correlation column shows Pearson correlation coefficient between method size and # of bug patterns. Defect density (size control) shows defect density of non-cloned code when (i) using the same number of non-cloned methods as cloned code methods (Equal # Methods); and (ii) using non-cloned methods whose LOC sums up to total cloned method LOC (Equal LOC

member of the same code clone group. This implies that the bug pattern was also duplicated during code cloning.

Category 2 - If the bug pattern is unique to the code clone group i.e., it is present in only one member of the group. This implies that the bug pattern was either introduced later (after code cloning) to one of the instances, or it was fixed in all the other instances after cloning.

Column 10 & 11 in Table 7.1 show number of duplicated and non-duplicated bug patterns for 29 (out of 31) projects respectively. In total, out of 339, we found that 254 code clone groups were classified in Category 1 and only 85 in Category 2.

75% of the FindBugs' bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones. Interpreting this result, while one may argue that cloning such a piece of code increased the total number of bug patterns, it is interesting and important to understand these results in the light of RQ2's results.

Most of these duplicated bugs again fall into coding and style category. On inspecting the code clone siblings with duplicated bug patterns, we found that it is very likely, that even without cloning, many developers would have written the code of same quality (i.e., with bug patterns present in them), if not worse. However, since bug patterns found in these categories are not very complex, a tool support to proactively assist developers make informed decision regarding cloning maybe very useful to maintain the code quality along with the advantages of rapid development using code cloning.

For example, we found that the two siblings of a clone group had identical code except hard coded integer values passed as parameter to the methods. Although this does not result in any serious threat to the program, it is definitely a bad practice because it breaks abstraction and increases the code size. In this case, an easy fix would be to refactor the two methods into one method by modifying the signature of any of these methods, so that it accepts an integer type as a parameter. Today, automated refactoring support for such scenarios often

leaves the burden of identifying the refactoring candidates on the developers and hence it is often perceived as an optional exercise. A more proactive approach, for example, in the above case, automatically detecting candidate methods for merging by detecting clones as soon as a method is completed by a developer, and then showing the identified FindBugs' bug pattern (hard-coded integer), will help developer to not only develop faster but also incrementally clean the code base. Thus we posit that making developers aware of such issues and assisting them with code refinement while they are copy pasting will blend better with the development activity and hence it is likely be more adopted.

75% of the FindBugs' bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones. As most of the bug patterns duplicated come from Coding & Style bug category, this finding has important implications on the part of tool designers

7.2.4 Conclusion (Study 1)

We conduct an empirical study of 31 open source Java projects to explore the relationship between code clones and a set of bug patterns reported by FindBugs and found that: (i) the defect density in cloned code is 3.7 times less than that of the rest of the code; (ii) 66% of the bug patterns associated with code clones are related to issues in coding style and practice, the two least problematic of the FindBugs' categories, while that number is 49% for non-cloned code; and (iii) 75% of the bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones.

7.3 Study 2. A Comparative Study of Software Quality Metrics in Java Cloned and Non-cloned Code

While the previous study looked at the relationship between code cloning and bug patterns, this study explores the relationship between code clones and 27 software quality metrics at a much larger scale. The study is conducted on 4,421 Java systems containing 1,486,882 methods.

I will briefly describe the key aspects of this study including research questions, dataset, how SourcererCC was used in this study, and finally summarize the key findings below. The complete study can be found at [22].

7.3.1 Research Questions

This study was motivated by the following research questions.

Research Question 1: *Are cloned methods less complex than the non-cloned methods in a project?*

Similar to the earlier study, this study was also conducted using intra-system method level clones only i.e., methods for which clones exist in the same project.

Complexity is an inherent problem to software systems, particularly when seen from software comprehension and maintenance aspects. The cost of software maintenance increases with the increase in the complexity of software [8]. If it is found that the complexity of the cloned methods is higher than the complexity of non-cloned methods, then it would mean that cloning contributes to increasing complexity of the system. Complexity is computed using 19 complexity metrics, which are shown in Table 7.2.

Research Question 2: *Are cloned methods more modular than the non-cloned methods in a project?*

Modularity is computed using five modularity metrics as shown in Table 7.2. If it is found that the cloned methods are less modular than the non-cloned methods, then it will be evidence against the practice of cloning, as the methods that are more modular are generally thought to be easier to maintain.

Research Question 3: *Are cloned methods more documented than the non-cloned methods in a system?*

A more documented code is easier to comprehend. If it is found that the cloned methods are less documented than the non-cloned methods, then it will contribute to the pool of evidences against the practice of cloning.

7.3.2 Dataset

The dataset used in this study consists of 4,421 Java projects hosted by Maven [88]. The comprehensive list of systems with their version information can be found at [22].

Figure 7.5 describes the size distribution of these projects. The X-axis represents the binned number of Java statements (NOS) and the Y-axis represents the percentage of projects in each bin.

7.3.3 Clone Detection

As in the case of Study 1, SourcererCC was used to compute method-level clones with a 0.7 similarity threshold.

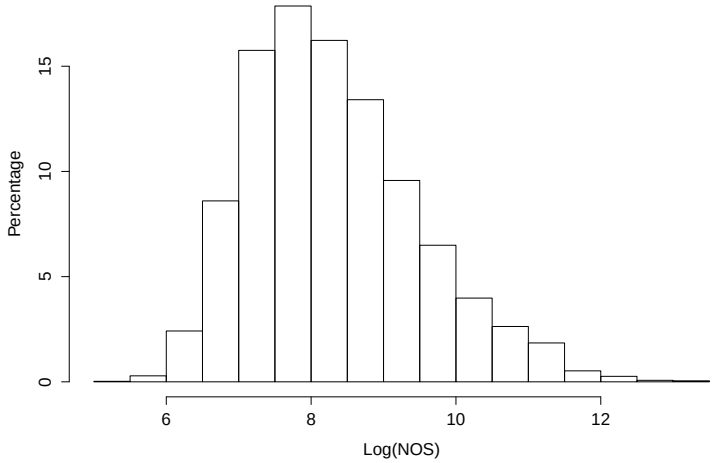

Figure 7.5: Size distribution of the projects. The X-axis represents the number of Java Statements in log scale (binned). The Y-axis shows the percentage of projects in each bin.

Precision. In order to measure the precision of SourcererCC for this study, 401 clone pairs were randomly chosen for manual inspection. This is a statistically significant sample with a 95% confidence level and a ±5% confidence interval. These clone pairs were inspected by 3 reviewers having more than 7 years of software development experience with at least 2 years of industrial experience.

Reviewer 1 found all 401 clone pairs to be true positives; reviewer 2 found 394 clone pairs as true positives and 7 as false positives; and reviewer 3 found 395 clone pairs to be true positives and 6 as false positives. After considering majority vote, only one clone pair was declared as a false positive and rest 400 were classified as true positives, resulting into 99.7% precision.

Distribution of method clones in the projects. In total 644,830 methods were found to have at least one clone whereas 842,052 methods had no clones in the entire corpus. Figure 7.6 shows the distribution of cloned methods in the projects. The X-axis represents the binned number of cloned methods in log scale. The Y-axis represents the percentage of projects. Figure 7.7 shows a similar graph for non-cloned methods.

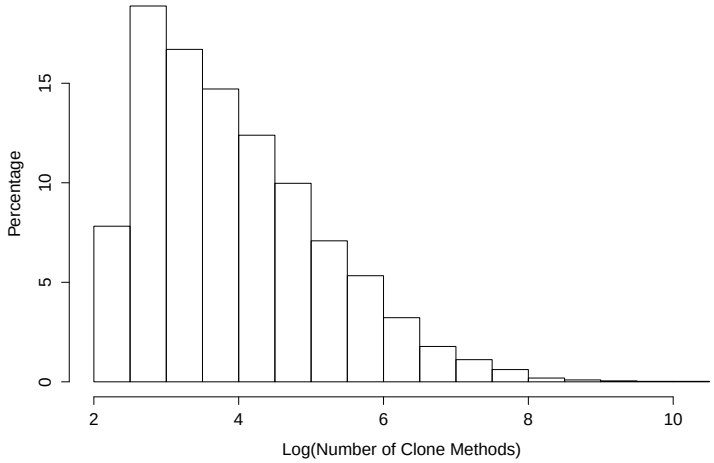

Figure 7.6: Distribution of subject systems measured using the number of cloned methods. The X-axis shows the binned number of clones in log scale. The Y-axis shows the percentage of systems in each bin.

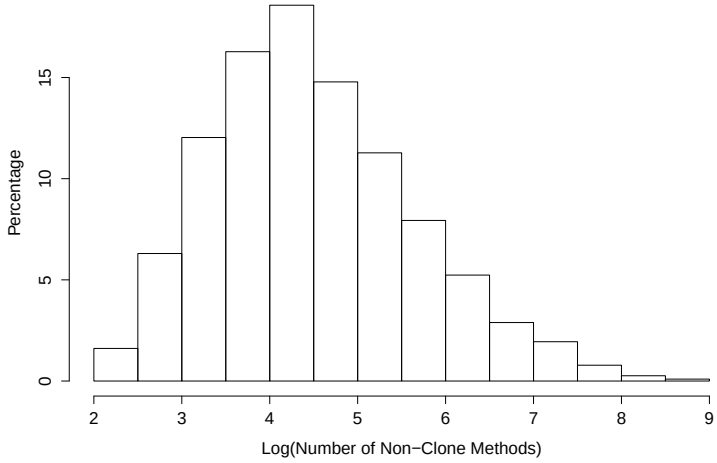

Figure 7.7: Distribution of subject systems measured using the number of non-cloned methods. The X-axis shows the binned number of non-cloned methods in log scale. The Y-axis shows the percentage of systems in each bin.

The median non whitespace lines of code (NLOC) for cloned and non-cloned methods is 11 and 14 respectively whereas the median number of statements (NOS) is 8 and 10 respectively.

7.3.4 Software Quality Metrics

For each method, 27 software quality metrics, (shown in Table 7.2) are computed across the three categories: (i) metrics assessing code complexity; (ii) metrics assessing modularity properties; and (iii) metrics assessing how well the code is documented

Finally, it was analyzed how these metric values differ for clones compared to rest of the code. Statistical analysis was used to seek answers to the research questions posed above.

7.3.5 Summary of the Results

This study found that statistically on an average 43% of the projects exhibit significant ($p<0.05$) difference in the metric values of cloned code and rest of the code in all of the three categories; with 90% of these projects showing at least small effect-size (r). Moreover, for 47% projects, complexity and modularity of cloned methods is found to be better than that of non cloned methods. However, non-cloned methods are found to be better documented than the cloned method i.e., non-cloned methods have higher number of code-comments than the cloned methods.

Following is a summary of the findings of three research questions from the study. Note that all values are statistically significant with p-value < 0.05 and effect-size > 0.1.

Complexity (RQ1): The complexity of cloned methods *differ* from the non cloned methods in 32% to 52% of the projects, and in 15% to 55.2% of the projects, the complexity of clones is significantly *better* than their counterparts.

Modularity (RQ2): The modularity of cloned methods *differ* from the non-cloned methods in 32% to 45% of the projects, and in 19% to 46% projects, the modularity of clones is significantly *better* than their counterparts.

Category		Name	Description
Code Complexity	↓	COMP	McCabes cyclomatic complexity
	↓	NOA	Number of arguments
	↓	VDEC	Number of variables declared
	↓	VREF	Number of variables referenced
	↓	NOS	Number of statements
	↓	NEXP	Number of expressions
	↓	MDN	Method, Maximum depth of nesting
	↓	HLTH	Halstead length of method
	↓	HVOC	Halstead vocabulary of method
	↓	HVOL	Halstead volume
	↓	HDIF	Halstead difficulty to implement a method
	↓	HEFF	Halstead effor to implement a method
	↓	TDN	Total depth of nesting
	↓	CAST	Number of class casts
	↓	LOOP	Number of loops (for,while)
	↓	NOPR	Total number of operators
	↓	NAND	Total number of operands
	↓	HBUG	Halstead prediction of number of bugs
	↓	NLOC	Number of lines of code
Modularity	↓	CREF	Number of classes referenced
	↓	XMET	External methods called by the method
	↓	LMET	Local methods called by the method
	↓	EXCR	Number of exceptions referenced by the method
	↓	EXCT	Number of exceptions thrown by the method
		MOD	Number of modifiers
Documentation	↑	NOC	Number of comments
	↑	NOCL	Number of comment lines

Table 7.2: Software Quality Metrics

Documentation (RQ3): The documentation of cloned methods differ from the non cloned methods in 43% of the projects, and in 36% to 42% projects, cloned methods are poorly documented than their counterparts.

7.3.6 Conclusion (Study 2)

This is the first study to empirically show that the characteristics of cloned methods differ from that of non-cloned methods at such a large scale (4,421 projects and 27 software metrics). These are observable characteristics which are measured using software quality metrics.

The work is ongoing and the authors are exploring ways to build classifier models in order to calculate a *cloneability index* of a method. The concept is similar to computing a maintainability index [95] of a method. The goal is to identify and monitor methods that show higher *cloneability index* and assess their impact on software maintenance.

7.4 Threats to Validity

In this section, we identify the threats to the validity of both the studies. While some threats may be pertinent to only one study, most of the them are applicable to both the studies.

Robustness of Clone Detection Technique. We classify methods as clones or nonclones. However, this classification is as accurate as the choice of our clone detection technique, tool, and configurations [125]. In order to mitigate this risk, we chose SourcererCC, our own tool to help us to choose the right configurations for better accuracy. Moreover we manually verified random samples and found false positives to be under 5%.

Granularity of Clones. Another threat could occur because of considering only method-level clones. We may miss overlapping clones or clones in class definitions. However, given this study is for Java projects, we believe that most of the clones representing logical blocks of program should be captured at a method level. Nonetheless, we acknowledge that this assumption might still impact the study.

False Positives in Bug Patterns. The imprecise nature of bug patterns reported by FindBugs is another threat to our validity. To address this issue, we only consider bug categories that have proven to be precise and actionable. Moreover, we exclude bug patterns with LOW severity. Nonetheless, it is very likely that we still face the issue of false positives. Moreover, although FindBugs detects a variety of bug patterns across various categories, there may be other bug patterns which are not detected. So the results of this study are to be interpreted only in the context of FindBugs' bug patterns.

Use of FindBugs in the Development Practice. It is possible that developers were already using FindBugs during development, and the warnings were fixed before release, and thus fewer post-release defects would be associated with warnings. This would further reduce the defect density and skew the analysis. We took a random sample of 20 projects,

and found no evidence of systematic use of FindBugs tool. We did this by going through the developer fixes, and examining the source history of these projects. In addition, we found no evidence in the email archives of any these projects suggesting a systematic adoption of FindBugs tool. These observations provide some mitigation to this particular threat.

Generalizability. The results of these studies are from open source Java systems, all medium to large size. We chose subject systems that exhibit variety in their type, size, and domain to minimize the impact of such factors on the observed phenomena. However, drawing general conclusions from empirical studies in software engineering is difficult because besides independent variables, the process depends on many relevant confounding variables [9].

For example, Mondal et al. [91] found that old clones are stable. Does stability of clones make them less bug prone? If yes, clone age could be one such confounding factor that needs to be controlled. Similarly, criticality of clones could play an important role i.e., are clones present in all the features or just not-so-critical features? This will validate that cloned code has fewer bug patterns not just because it does not implement a critical feature of the system, but because of its peculiarity. This can be done by preserving the homogeneity of cloned and non-cloned methods, which is currently not accounted for.

Thus, in the presence of such confounding factors, we cannot assume *a priori* that the results of the study generalize beyond the setting for which it was conducted. However, the overall results of this study showed several commonalities across a wide range of systems and indicate that the results hold for more than just the studied systems.

7.5 Reproducibility

While our results appear to be statistically significant, we urge caution in extending the findings to other languages. Comparisons such as these are key to promoting software engineering discipline in understanding code clone properties and we invite others to use our dataset for further experiments. We have made available all the necessary artifacts including project sources, detailed steps to run tools and produce the raw data, analysis steps to produce the statistical results to verify the claims for both the studies.

1. A Comparative Study of BugPatterns in Java Cloned and Non-cloned Code: `http://mondego.ics.uci.edu/projects/bugpatternsinclones`

2. A Comparative Study of Software Quality Metrics in Java Cloned and Non-cloned Code: `http://mondego.ics.uci.edu/projects/clone-metrics`

7.6 Chapter Summary

This chapter presented two empirical studies conducted using SourcererCC to demonstrate the effectiveness of SourcererCC for empirical clone research.

The goal of these studies is to explore the relationship between code clones and various quality attributes (software quality metrics and bug patterns). These studies are not the final word on the issue, but it is one more piece of evidence that in the practice of software development, clones do not seem to be as bad as they have been thought to be. While both these studies does not unveil any explanation for these findings, results from other, more qualitative studies indicate that the developers use copy-and-paste intentionally and wisely, which may explain the quantitative observations of these studies.

Like all complex problems, the issue of code cloning being bad or not will only be fully

understood by looking at it from several angles and with several methodologies. But if clones are not as bad as we thought they were, this leads to interesting new avenues of exploration for tools that help *manage* clones rather than *eliminating* them. Such tools and techniques can help developers take advantage of rapid development using cloning and also manage clones effectively to avoid degrading the quality of code due to cloning.

Overall, these research results suggest that the practice of code cloning in Java, and possibly in all other object-oriented languages, needs to be given serious consideration on the part of tool designers.

Chapter 8

Conclusions and Discussion

8.1 Dissertation Summary

This dissertation focused on the topic of large-scale code cloning. While several techniques have been proposed for clone detection over many years, accuracy and scalability of clone detection tools and techniques still remains an active area of research. Specifically, there is a marked lack of clone detectors that scale to large systems or repositories, particularly for detecting near-miss clones where significant editing activities may take place in the cloned code. Furthermore, with the amount of source code increasing steadily, large-scale clone detection has become an even greater necessity. These large code bases and repositories of projects have led to several new use cases of clone detection, including mining library candidates, detecting similar mobile applications, detection of license violation, reverse engineering product lines, finding the provenance of a component, and code search.

To that end, this dissertation presented SourcererCC, a token-based accurate near-miss clone detection tool, which uses an optimized partial index and filtering heuristics to achieve large-scale clone detection on a standard workstation.

SourcererCC's scalability is demonstrated with IJaDataset, a large inter-project repository containing 25,000 open-source Java projects, and 250 MLOC. We measure SourcererCC's recall using two state-of-the-art clone benchmarks, the Mutation Framework and BigCloneBench. We found that SourcererCC is competitive with even the best of the state-of-the-art Type-3 clone detectors. To measure precision, we conducted blind experiments using five reviewers and statistically significant samples of clones detected by the tools to measure precision. We found SourcererCC to have a high precision (86%). We believe that SourcererCC can be an excellent tool for various modern use cases that require reliable, complete, fast, and scalable clone detection.

This dissertation made five major contributions in the field of code clone detection research. The specific contributions can be summarized as follows:

1. **The SourcererCC Tool (Chapter 3)** - The SourcererCC tool represents an important contribution in the area of code clone detection research. Large-scale empirical studies on code cloning depend on the quality and scalability of clone detection tools. SourcererCC provides an infrastructure to enable high quality empirical clone research. Chapter 3 presented a detailed description of the SourcererCC tool, including information about how to use it.

2. **Tool Comparison Benchmarks (Chapter 4)** - The scalability and accuracy (precision and recall) of SourcererCC is evaluated against four state-of-the-art clone detectors using multiple datasets. The results of these experiments can serve as benchmarks for future research in the area of clone detection.

3. **The SourcererCC-D Tool (Chapter 5)** - The SourcererCC-D tool is a distributed version of SourcererCC that can be easily deployed on Amazon Web Service or machines with multiple processors. To the best of our knowledge, it is the first publicly available distributed clone detector. Chapter 5 presented a detailed description of the tool,

including information about how to use it.

4. **The SourcererCC-I Tool (Chapter 6)** - While there are several existing clone detection tools that are beneficial in the analysis and investigation of code clones and their evolution, they fail to provide necessary clone management support for clone-aware development activities, as they are not integrated with the development environment. Chapter 6 presents SourcererCC-I, an Eclipse plug-in based on top of SourcererCC, that instantaneously reports intra- & inter-project method level clones in Java projects. SourcererCC-I is designed with features to support clone-aware development and maintenance activities.

5. **Empirical Studies on the Relationship of Code Clones and Quality Attributes (Chapter 7)** - Due to the challenges discussed earlier, there is a lack of studies on understanding the relationship between code cloning and quality attributes. Chapter 7 presents two empirical studies to explore the relationship between code clones and (i) bug patterns and (ii) software quality metrics. We found that there is a positive differentiation of cloned code with respect to the rest of the code when using bug patterns and software metrics. While a qualitative analysis of the findings is certainly encouraged, overall, these research results suggest that the practice of code cloning in Java, and possibly in all other object-oriented languages, needs to be given serious consideration on the part of tool designers. These studies also demonstrate how SourcererCC can be effectively used for large-scale empirical clone research.

8.2 The Surprising Effectiveness of the Bag-of-tokens model and Overlap Similarity Measure in Clone Detection

The core of SourcererCCs algorithm is based on two important but simple concepts:

(i) Bag-of-tokens model to represent code fragments; and (ii) overlap similarity measure to compute similarity between code fragments.

SourcererCC represents a code block using a bag-of-tokens model where tokens are assumed to appear independently of one another and their order is irrelevant. The idea is to transform code blocks in a form that enables SourcererCC to detect clones that have different syntax but similar meaning. Moreover, this representation also filters out code blocks with specified structure patterns. Since SourcererCC matches tokens and not sequences or structures, it has a high tolerance to minor modifications, making SourcererCC effective in detecting Type-3 clones, including clones where statements are swapped, added, and/or deleted.

The overlap similarity measure simply computes the intersection between the code fragments by counting the number of tokens shared between them. The intuition here is simple. If two code fragments have many tokens in common then they are likely to be similar to some degree.

It is interesting to note that such a simple strategy could prove to be so effective in a complex software engineering task of identifying code clones. We believe that one of the reasons for its effectiveness is the vocabulary developers use to write source code. While programming languages in theory are complex and powerful, the programs that real people write are mostly simple and rather repetitive and similar [47]. This similarity is manifested in the source code in the form of tokens, and particularly in identifiers. In source code,

identifiers (e.g. names of variables, methods, classes, parameters, or attributes) account for approximately more than 70% of the linguistic information [33]. Many researchers have concluded that identifiers reflect the semantics and the role of the named entities they are intended to label [18, 79, 43]. Therefore, code fragments having similar semantics are likely to have similarity in their identifiers. Furthermore, oftentimes, during copy-paste-modify practice, developers preserve identifier names as they reflect the underlying functionality of the code that is copied. They seem to be aware of the fact that different names used for the same concept or even identical names used for different concepts reflect misunderstandings and foster further misconceptions [33]. As a result, while copied fragments are edited to adapt to the context in which they are copied, they often have enough syntactical similarity associated with the original fragment. This similarity is effectively captured by the bag-of-tokens model in conjunction with the overlap similarity measure.

Of course, there are scenarios when programmers may deliberately obfuscate code to conceal its purpose (security through obscurity) or its logic, in order to prevent tampering, deter reverse engineering, hide plagiarism, or as a puzzle or recreational challenge for someone reading the source code. The simple bag-of-tokens model of SourcererCC may not be effective in detecting clones in such cases. Other tools like Deckard that rely on AST, or NiCad that uses heavy normalizations, may be effective under such scenarios.

8.3 Lessons Learned During SourcererCC's Development

In this section, I will try to summarize the lessons that I learned during the design, development and testing of SourcererCC. While these lessons are not new and are already well-known in the field, my dissertation work has given me an opportunity to directly experience and

reflect upon them.

Everything breaks at scale. One of my key lessons during SourcererCC's development can be aptly described in a phrase - "Everything breaks at scale, so expect the unexpected". We realized that at scale, we cannot test for every error. As a result, we used assertions and exception handlers for things that can't happen. We added diagnostic code, logging and tracing to help explain what is going on at run-time, especially when we ran into problems during development. The philosophy - if this failed, look for what else can fail - played a very important role during SourcererCC's development.

Fault Tolerance. During the initial stages of the development, SourcererCC crashed at times while running on large datasets after several hours of execution due to unexpected reasons. Since SourcererCC did not have mechanism to preserve its execution state during that time, such failures resulted in a loss of several hours of computation time and effort. Not to mention the frustration that comes along. Based on these experiences, we realized that SourcererCC's exception handler must preserve its state of the execution (i.e., keeping track of how much data is already processed), so when interrupted, SourcererCC's execution can resume correctly from the point of failure at a later time. The necessity of logging how much data is processed by the tool is an important lesson that we learned the hard way.

Memory Leaks. SourcererCC is written in Java programming language which has its own garbage collection mechanism. However, we encountered bugs related to memory leaks while testing SourcererCC on large datasets. What I learned from debugging memory leak issues is that while there are simple solutions to detect and deal with memory leaks (e.g., logging the size of your data structures when you modify them, and then searching the logs for data structures that grow beyond a reasonable size), a tool is undoubtedly a big help. In the absence of the right tools, debugging such issues could take unreasonable time and effort. We were able to resolve these issues much faster using open source tools like VisualVM[1] and

[1]https://visualvm.java.net/

Profiler4J[2].

The problem could be in the data too. Oftentimes when we noticed anomalies in the execution of SourcererCC, we thought that the issue would be in the code. However, it was not unusual to find issues with either the input data or our assumptions about the input data. As a result, we realized that it is always useful to check for data consistency and integrity even before any experimentation.

Tuning parameters to optimize SourcererCC's performance. SourcererCC has few parameters (e.g., similarity threshold, tokenization startegies, minimum size threshold of a code block) that had to be tuned to optimize for accuracy, scalability, and efficiency. This resulted in countless experiments, and keeping track of these experiments and their settings posed a severe challenge.

To do this exercise systematically, we adopted the following process that indeed turned out to be very effective.

We created a smaller dataset for parameter tuning experiments. Apart from the smaller size, this dataset had characteristics very similar to the large datasets on which SourcererCC is intended to be used. Executing SourcererCC on a smaller dataset took less time, thus giving us more freedom to experiment.

In order to better keep track of SourcererCC's performance on different parameter configurations, we created a SourcererCC revision (using Git) for each configuration of parameters. This not only enabled us to run several experiments in parallel, but also helped to easily switch back-and-forth across various parameter configurations.

To summarize, creating SourcererCC's revisions for various parameter configurations and running them in parallel on a smaller dataset greatly reduced the turn around time for

[2]http://profiler4j.sourceforge.net/

performing experiments to tune SourcererCC.

8.4 Going Forward

Code Clone detection research has come a long way in the last couple of decades. We conclude by identifying some of the relevant areas that might shape the future research in this field.

There are many tools available for clone detection. In contrast, there are relatively few tools that help in removing or effectively managing clones. Identifying various means of eliminating harmful clones through automated tool support is an interesting venue to explore in the future.

Large-scale clone detection is often faced with the challenge of how to make sense of the large data produced by the clone detection tools. Visual and interactive representations of the output to reinforce human cognition and produce actionable insight is another useful direction for the future.

The utility of clone detection is not just limited to source code. Clone detection in other software artifacts, including models, bug-reports, requirement documents, and binaries, is turning out to be a necessity for several use cases. For example, the ability to detect clones in software binaries is necessary for effectively detecting Malwares and License Infringement. Therefore, extending code clone detection research to other software artifacts is a promising area for the future.

Clone research should also focus on clone management by: (i) identifying and prioritizing the clones that are of interest to the developers for a given task; (ii) helping developers proactively assess the negative consequence of cloning; and (iii) categorizing clones as harmful

and harmless after detection.

With the several new use cases of clone detection emerging, a reorientation of research focus towards application-oriented clone detection might be useful. In many cases, state-of-the-art clone detection tools do not behave well for these specific use cases. These observations point to the new research opportunities to enhance clone detection technologies. Moreover, use case specific benchmarking to evaluate various tools and techniques might be another area to focus on in the future.

Bibliography

[1] Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165 – 1199, 2013.

[2] Cloc: Count lines of code. `http://cloc.sourceforge.net`, 2015.

[3] Ambient Software Evoluton Group. IJaDataset 2.0. `http://secold.org/projects/seclone`, January 2013.

[4] N. Ayewah and W. Pugh. Using findbugs on production software. In *In OOPSLA 07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 805–806. ACM, 2007.

[5] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 24–49, 1992.

[6] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of Working Conference on Reverse Engineering*, 1995.

[7] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.

[8] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, 1993.

[9] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25(4):456–473, 1999.

[10] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *In Proc. of the Intl Conf. on Software Engineering*, pages 451–459, 2005.

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, page 368. IEEE Computer Society, 1998.

[12] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[13] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, Sept 2007.

[14] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[15] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the the Seventh joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.

[16] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying cross cutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[17] E. Burd and M. Munro. Investigating the maintenance implications of the replication of code. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 322–329, Oct 1997.

[18] C. Caprile and P. Tonella. Nomen est omen: analyzing the language of function identifiers. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 112–122, Oct 1999.

[19] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère. An empirical assessment of bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 20:1–20:10, New York, NY, USA, 2015. ACM.

[20] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 5–, Washington, DC, USA, 2006. IEEE Computer Society.

[21] 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 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[22] Clone metrics. http://mondego.ics.uci.edu/projects/clonemetrics/.

[23] P. Clough and D. O. I. Studies. Old and new challenges in automatic plagiarism detection. In *National Plagiarism Advisory Service, 2003; http://ir.shef.ac.uk/cloughie/index.html*, pages 391–407, 2003.

[24] J. Cordy. The txl programming language. `http://www.txl.ca/`.

[25] J. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proceedings of International Conference on Program Comprehension*, pages 196–205, 2003.

[26] J. Cordy and C. Roy. The nicad clone detector. In *Proceedings of ICPC*, 2011.

[27] J. R. Cordy. Comprehending reality-practical barriers to industrial adoption of software maintenance automation. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 196–205. IEEE, 2003.

[28] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 219–220, Washington, DC, USA, 2011. IEEE Computer Society.

[29] Coverity scan. http://scan.coverity.com/.

[30] Darpa muse. http://www.darpa.mil/program/mining-and-understanding-software-enclaves.

[31] J. Davies, D. German, M. Godfrey, and A. Hindle. Software Bertillonage: finding the provenance of an entity. In *Proceedings of MSR*, 2011.

[32] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*. IEEE Computer Society, 2009.

[33] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, Sept. 2006.

[34] Developer survey. http://tinyurl.com/oezms7d.

[35] E. Duala-Ekoko and M. P. Robillard. Clonetracker: tool support for code clone management. In *Proceedings of the 30th international conference on Software engineering*, pages 843–846. ACM, 2008.

[36] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 109. IEEE Computer Society, 1999.

[37] Findbugs. http://findbugs.sourceforge.net/.

[38] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[39] D. M. German, M. D. Penta, Y. gal Guhneuc, and G. Antoniol. Code siblings: technical and legal implications of copying code between applications. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, 2009.

[40] N. Gode and R. Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 219–228, March 2009.

[41] N. Gode and R. Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 219–228. IEEE, 2009.

[42] N. Gode and R. Koschke. Incremental clone detection. In *Proceedings of CSMR*, 2009.

[43] L. Guerrouj. Normalizing source code vocabulary to support program comprehension and software quality. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1385–1388, May 2013.

[44] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *Proceedings of Working Conference on Reverse Engineering*, pages 357–366, 2012.

[45] I. Herraiz, E. Shihab, T. H. Nguyen, and A. E. Hassan. Impact of installation counts on perceived quality: A case study on Debian. In *Proceedings of the 18th Working Conference on Reverse Engineering*. IEEE Computer Society, 2011.

[46] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[47] 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, Piscataway, NJ, USA, 2012. IEEE Press.

[48] D. Hou, P. Jablonski, and F. Jacob. Cnp: Towards an environment for the proactive management of copy-and-paste programming. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 238–242. IEEE, 2009.

[49] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[50] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection:incremental, distributed, scalable. In *Proceedings of ICSM*, 2010.

[51] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 387–391, Oct 2012.

[52] H. T. Jankowitz. Detecting plagiarism in student pascal programs. *Comput. J.*, 31(1):1–8, Feb. 1988.

[53] Y. Jia, B. Binkley, M. Harman, J. Krinke, and M. Matsushita. A proposed approach to fast and precise clone detection. In *Proceedings of IWSC*, 2009.

[54] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE*, 2007.

[55] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 96–105, May 2007.

[56] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, pages 171–183, Toronto, Ontario, Canada, 1993. IBM Press.

[57] J. H. Johnson. Substring matching for clone detection and change tracking. In *International Conference on Software Maintanence*, pages 120–126, 1994.

[58] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective a workbench for clone detection research. In *Proceedings of ICSE*, 2009.

[59] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. do code clones matter?. In *Proceedings of ICSE*, pages 485–495, 2009.

[60] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

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

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

[63] C. Kapser and M. Godfrey. cloning considered harmful considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[64] 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. volume 0, pages 313–314, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[65] I. Keivanloo, J. Rilling, and P. Charland. Internet-scale real-time code clone search via multi-level indexing. In *Proceedings of WCRE*, 2011.

[66] I. Keivanloo, C. Roy, J. Rilling, and P. Charland. Shuffling and randomization for scalable source code clone detection. In *Proceedings of IWSC*, 2012.

[67] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 58–64, New York, NY, USA, 2006. ACM.

[68] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of FSE*, 2005.

[69] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer-Verlag, 2001.

[70] R. Komondoor and S. Horwitz. Effective automatic procedure extraction. In *Proceedings of the International Workshop on Program Comprehension*, pages 40–56. Springer-Verlag, 2003.

[71] R. Koschke. Survey of research on software clones. In *Proceedings of Duplication, Redundancy, and Similarity in Software*, 2007.

[72] R. Koschke. Identifying and removing software clones. pages 15–36, 2008.

[73] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *Proceedings of CSMR*, pages 309–318, 2012.

[74] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, 2006.

[75] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[76] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301. IEEE Computer Society, 2001.

[77] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Working Conference on Reverse Engineering*, pages 170–178, 2007.

[78] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 314–321. IEEE, 1997.

[79] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 3–12, June 2006.

[80] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 167–176, New York, NY, USA, 2010. ACM.

[81] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[82] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, volume 4, pages 289–302, 2004.

[83] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of ICSE*, 2007.

[84] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Mining Software Repositories*, pages 18–22, 2007.

[85] K. M., N. C., N. J., and L. W. *Applied Linear Statistical Models*. Sage Publications.

[86] U. Manber. Finding similar files in a large file system. In *USENIX WINTER 1994 TECHNICAL CONFERENCE*, pages 1–10, 1994.

[87] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of International Conference on Software Engineering*, 2003.

[88] A. Maven. http://maven.apache.org/.

[89] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*. IEEE, 1996.

[90] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 244–253, Nov 1996.

[91] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1227–1234, New York, NY, USA, 2012. ACM.

[92] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 87–, Washington, DC, USA, 2002. IEEE Computer Society.

[93] K. S. Muhammad Asaduzzaman, C. K. Roy and M. D. Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 2013.

[94] 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 International Conference on Software Maintenance*. IEEE Computer Society, 2009.

[95] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[96] J. Ossher, H. Sajnani, and C. V. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *ICSM*. IEEE, 2011.

[97] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[98] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

[99] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing Static Bug Finders and Statistical Prediction. In *Proceedings of the International Conference on Software Engineering*, Hyderabad, India, 2014. ACM.

[100] Rajapakse, D. C., and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of International Conference on Software Engineering*, pages 116–126, 2007.

[101] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Bern, 2005.

[102] C. Roy and J. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 157–166, April 2009.

[103] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 18–33, Feb 2014.

[104] C. K. Roy and J. R. Cordy. A survey on software clone detection research. (TR 2007-541), 2007. 115 pp.

[105] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.

[106] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. *J. Softw. Maint. Evol.*, 22(3):165–189, Apr. 2010.

[107] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program*, pages 470–495, 2009.

[108] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. of Comput. Program.*, pages 577–591, 2009.

[109] 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.

[110] H. Sajnani, V. Saini, and C. V. Lopes. A comparative study of bug patterns in java cloned and non-cloned code. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 21–30. IEEE, 2014.

[111] H. Sajnani, V. Saini, J. Ossher, and C. Lopes. Is popularity a measure of its quality? an analysis of maven components. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution(To appear in ICSME 2014)*. IEEE Computer Society, 2014.

[112] M. Shomrat and Y. Feldman. Detecting refactored clones. In G. Castagna, editor, *ECOOP 2013 Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526. Springer Berlin Heidelberg, 2013.

[113] T. Suresh, C. Luigi, A. Lerina, and D. P. Massimiliano. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[114] 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 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 476–480, Washington, DC, USA, 2014. IEEE Computer Society.

[115] J. Svajlenko, I. Keivanloo, and C. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *Software Clones (IWSC), 2013 7th International Workshop on*, pages 16–22, May 2013.

[116] J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using classical detectors: an exploratory study. *Journal of Software: Evolution and Process*, 27(6):430–464, 2015.

[117] J. Svajlenko and C. K. Roy. Evaluating modern clone detection tools. In *ICSME*, 2014. 10 pp.

[118] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, page 10, 2015.

[119] J. Svajlenko, C. K. Roy, and J. R. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pages 8–9, 2013.

[120] Tcs banking case study. http://www.tcs.com/sitecollectiondocuments/case

[121] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180, 2004.

[122] S. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of Working Conference on Reverse Engineering*, 2011.

[123] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE*, pages 285–294, 2003.

[124] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *ESEC/FSE*, pages 455–465, 2013.

[125] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 455–465, New York, NY, USA, 2013. ACM.

[126] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 131–140, New York, NY, USA, 2008. ACM.

[127] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo. An empirical study on the fault-proneness of clone migration in clone genealogies. In *Proc. of CSMR-WCRE*, pages 94–103. IEEE, 2014.

[128] Y. Zhang, R. Jin, and Z.-H. Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010.

[129] M. F. Zibran and C. K. Roy. Ide-based real-time focused search for near-miss clones. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1235–1242, New York, NY, USA, 2012. ACM.

[130] G. K. Zipf. Selective Studies and the Principle of Relative Frequency in Language.

Appendices

A Subject Systems

Table A.1 describes the details of 35 open source Apache Java projects referred in Chapter 3 and Chapter 4. These projects are of varied size and span across various domains including search and database systems, server systems, distributed systems, machine learning and natural language processing libraries, network systems, etc. Most of these subject systems are highly popular in their respective domain. The details include project name, size of the project in KLOC, the number of methods in each project and the number of clone pairs found using SourcererCC at 0.7 similarity threshold.

Subject System	Size (KLOC)	# Methods	# Clones
j2sdk1.4.0-javax-swing	102.836	12,174	621
eclipse-jdtcore	98.169	8,716	1,720
jython	211.905	20,284	2,843
jfreechart	93.46	7,783	1,317
ant	86.438	7,695	192
hadoop-hdfs	70.411	6,004	3,763
stanford-nlp	210.233	9,853	316
cloud9	56.766	4,720	2,074
hibernate	72.409	9,286	205
tomcat-catalina	73.673	5,531	329
hadoop-mapred	64.023	5,318	216
mahout-core	53.366	4,013	181
xerces	76.185	6,122	398
berkeleyparser	57.905	3,945	453
rhino	54.722	3,174	129
pmd	60.06	5,358	528
synapse-core	41.612	3,146	166
substance	47.361	3,049	198
poi	47.804	4,897	1,129
pig	84.77	5,910	520
mason	35.931	2,858	494
eclipse-ant	16.106	1,907	34
postgresql	23.514	1,989	251
struts	24.799	2,445	132
httpclient	18.022	1,383	72
log4j	20.611	1596	286
cglib	13.668	2,135	16
dom4j	17.854	2,396	211
nutch	12.243	857	27
lucene	15.67	1,301	56
uima-core	11.942	1,477	62
netbeans-javadoc	9.579	959	59
cocoon	10.387	751	16
pdfbox	13.936	860	31
commons-io	8.673	859	53
junit	6.728	955	8

Table A.1: Performance of the filtering technique by comparing the time taken and total number of comparisons done to detect clones with and without filtering technique used

B Running SourcererCC-D Using Amazon Web Services (AWS)

The three main components of SourcererCC-D for running on AWS's infrastructure are: (i) StarCluster utility to launch AWS clusters; (ii) SourcererCC; and (iii) Shell scripts to distribute and automate the clone detection process. However, for running SourcererCC-D on any in-house cluster, we simply need (ii) and (iii).

Figure B.1 shows the implementation architecture of SourcererCC-D to run on AWS. In order to deploy SourcererCC-D on AWS, we first need to install StarCluster. StarCluster is an open source cluster-computing toolkit for Amazons Elastic Compute Cloud (EC2). It has been designed to automate and simplify the process of building, configuring, and managing clusters of virtual machines on Amazons EC2 cloud. It allows to easily create a cluster computing environment in the cloud suited for distributed and parallel computing applications and systems. The instructions to download and setup StarCluster is available at its webpage: `http://star.mit.edu/cluster/`.

After installing the StarCluster, user needs to update its configuration file using user's AWS credentials. Once configured, one simply needs to run a single command to launch SourcererCC-D on AWS. Below we describe how this automation is achieved using shell scripts.

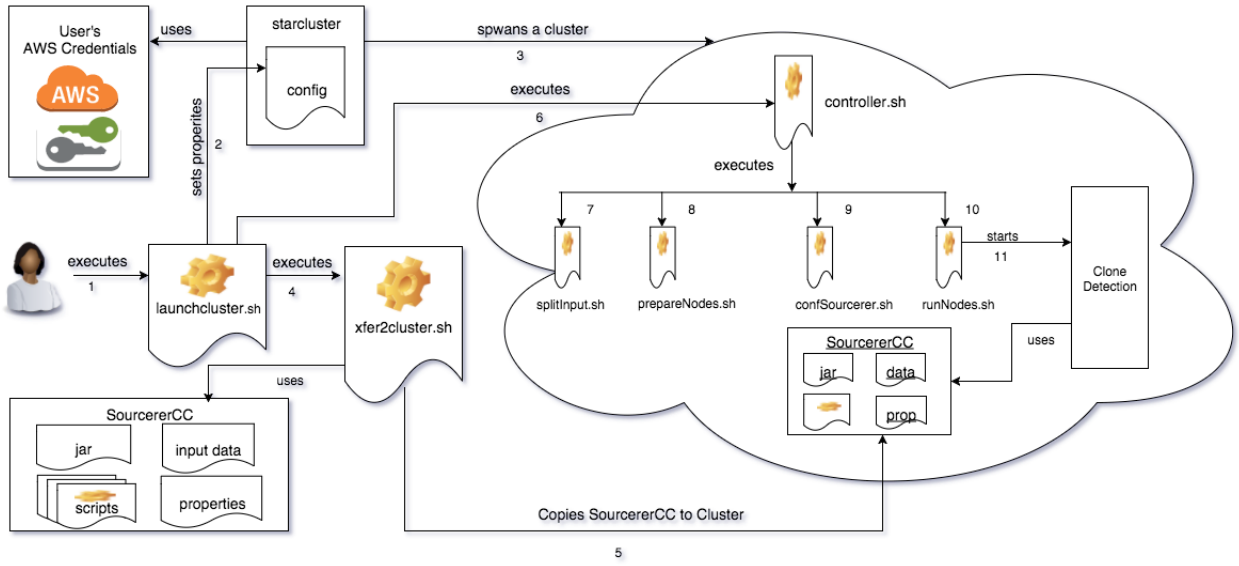

Figure B.1: SourcererCC-D's Implementation Architecture

1. launchcluster.sh

This is the main script that is responsible for: (i) launching the cluster; (ii) transferring the *data* on the cluster (using xfer2cluster.sh) and (iii) splitting the parsed input and distributing the jobs to the worker nodes (using controller.sh).

While it may seem like a series of complex operations, all this happens internally and user is agnostic of these operations. A user needs to only prepare a *data* folder on his local machine that contains (i) input files generated by the parser; (ii) SourcererCC library; (iii) Automation scripts. After that, it is simply a single command to run SourcererCC-D to detect clones.

For example, in order to launch SourcererCC-D on a cluster of 10 working nodes, user can simply execute the following command:

```
./launchcluster 11
```

Here 11 specifies that a user wants to launch a cluster with 1 master node and 10 working nodes. After the launch is successful, the script internally invokes *xfer2cluster.sh*.

2. xfer2cluster.sh

This script transfers the *data* folder prepared by the user on the cluster. After the transfer is done, *controller.sh* is invoked on the cluster.

3. controller.sh

The script invokes *splitquery.sh* to split the input files into N almost equally sized query files, where $N$is the number of worker nodes in the cluster. Next it assigns every worker node in the cluster with exactly one query file. In the end, it executes *runnodes.sh* to create indexes and detect clones.

4. runnodes.sh

This script commands the master node to build a global index of the entire corpus using SourcererCC's Indexer. The constructed global index will be stored on a disk shared by all the worker nodes. After the global index is constructed, the script triggers each worker to launch the search process for its local input file using SourcererCC's Searcher. Every worker node reports result in its own output folder on the shared disk. Once all the workers have finished executing, all the clones in the corpus have been found.

C Experience Report on Using AWS

In this section, we describe our experience in using AWS for our experiments. While AWS has a large number of physical servers under management it does not rent them per-se, rather only access to these servers in the form of virtual machines is available. The types of virtual machines are limited to a small list so as to make choosing an instance relatively easy. The advantage is that all these virtual machines come with pre-installed software like Java, Hadoop or other necessary tools. And although AWS claims that the virtual machines yield roughly the same performance in terms of compute, we noticed that even same instance type could run on very different underlying hardware platforms impacting performance. Hence, performance numbers may vary even with same configurations. We recommend running experiments multiple times and report the average. Also, it is advisable to use this service to find relative performance numbers e.g., properties like scale-up and speed-up, but may not be appropriate in case where absolute numbers are of importance.

At first, when we started developing our application, we had no idea if we were going to use AWS. Later, we surveyed all the options and decided to deploy our application to AWS. Since the application was not tailored for AWS, whenever the deployment crashed, we had difficulty in debugging. Later, we realized that AWS provides a toolkit for Eclipse, which is a plug-in for the Eclipse Java IDE that makes it easier to develop, deploy, and debug Java applications using Amazon Web Services. After installing the toolkit, the process was much smoother. So if you already know that you might use AWS to deploy your application, we recommend using the toolkit as early as possible to avoid any migration cost later.

We also realized that although Amazon provides a web management console, it is best to use Amazon's command line ruby client for running and monitoring experiments. The web console experiences a lag and fails to update the information regarding instance allocation and usage on time. We made decisions to terminate the process several times based on the

stale information available through the web console. On the other hand, the ruby client is a quick command line client for monitoring the jobs efficiently.

While deploying an application on AWS, it is important to spend time thinking about the type of instances that would be best for your application. AWS offers wide variety of instances specially optimized for computation, storage, memory, GPU, etc. The performance can vary widely based on the choice of instances made. Hence, we recommend considering this as part of your design process itself. For example, in our case, since each machine had a limited workload, we rented smaller machines with reasonable memory and storage and did not opt for expensive machines with higher RAM and SSDs.

Overall, we found that although AWS has certain limitations, it proves to be an excellent affordable service for large scale experiments involving tens and hundreds of machines.

D Cost of Running the Experiments Using AWS

We rented a total of 62 Amazon $C3.2$ large instance machines (31 to measure the speed-up and 31 to measure the scale-up) for a total of 268 hours for both the experiments. The price of an instance is \$0.46 per hour. Thus, we spent a total of \$123.28 for a single run of the experiments. Since we ran the experiments twice to account for variation in the results, the total cost incurred is \$246.56.