Language-agnostic Incremental Code Clone Detection

Master's Thesis

Stavrangelos Gamvrinos

Language-agnostic Incremental Code Clone Detection

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Stavrangelos Gamvrinos
born in Athens, Greece

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Software Improvement Group
Fred. Roeskestraat 115
Amsterdam, the Netherlands
www.softwareimprovementgroup.com

©2020 Stavrangelos Gamvrinos. All rights reserved.

Language-agnostic Incremental Code Clone Detection

Author: Stavrangelos Gamvrinos
Student id: 4771583
Email: s.gamvrinos@student.tudelft.nl

Abstract

Code duplication is a form of technical debt frequently observed in software systems. Its existence negatively affects the maintainability of a system in numerous ways. In order to tackle the issues that come with it, various automated clone detection techniques have been proposed throughout the years. However, the vast majority of them operate using the entire codebase as input, resulting in redundant calculations and undesirable delays when this process is repeated for every new revision of a project. On the other hand, newer incremental techniques address this by storing intermediate information that can be reused across revisions. However, all these approaches are language-specific, utilizing language parsers to generate more sophisticated source code representations, in an attempt to detect more complex types of clones. As a result, less popular languages, for which finding or building a parser is challenging, are unfortunately not supported.

In this study we propose LIICD, a language-agnostic incremental clone detector, capable of detecting exact-match clones. We assess its performance and compare it with a state-of-the-art commercial-grade detector, found within the Software Improvement Group (SIG). Furthermore, we use a similarity estimation technique called Locality Sensitive Hashing (LSH) in an attempt to extend and improve the original approach. Our experiments result in some interesting findings. Firstly, the proposed incremental detector is very efficient and able to scale well for larger codebases. Additionally, it provides a significant improvement compared to a non-incremental commercial-grade detector. Lastly, our LSH-based extension proves to have difficulties matching our original approach's performance. However, future suggestions indicate how the potential of the technique can be further investigated.

Thesis Committee:

University supervisor:	Prof. Dr. A. van Deursen, TU Delft
Committee Member:	Assistant Prof. Dr. M. Aniche, TU Delft
Committee Member:	Assistant Prof. Dr. C.B. Poulsen, TU Delft
Company supervisor:	Marco di Biase, SIG

Preface

This study marks the end of my masters studies at TU Delft. Retrospectively, mixed feelings emerge when looking back at these two years. On the one hand, the curriculum was less technical and much more research-oriented than I would desire, leading me to question my decision to pursue it. On the other hand, I learned a lot of things and was able to discover interests and future aspirations that I would probably have otherwise missed. In the meantime, I also met great people who supported me throughout this journey.

With the completion of this work, I would first like to thank my university supervisor, Arie van Deursen, for his high-level guidance, feedback and contribution regarding the direction of this study. Furthermore, I would like to thank Marco di Biase, my daily supervisor, whose help played a determining role in the completion of this study. Not once did he complain about all the questions and extra work that I sometimes had to put on him.

From SIG, my gratitude goes to everyone within the research team. I am glad I was given the opportunity to be hosted at the company's premises and was able to meet great people with great minds. Also, kudos to everyone outside the research team, who helped me one way or the other during the past eight months.

Moreover, I would like to thank my friends for all the memories and support during the tough, in terms of workload, times during my studies. Last but not least, all this would not be possible without the immense support of my family, whose help I deeply appreciate.

Stavrangelos Gamvrinos
Delft, the Netherlands
July 20, 2020

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Research Questions . . . 3
- 1.3 Scope . . . 4
- 1.4 Contributions . . . 5
- 1.5 Thesis Outline . . . 6

2 Background & Related Work 7
- 2.1 Clone Detection Basics . . . 7
- 2.2 Clone Detection Challenges . . . 10
- 2.3 Clone Detection Techniques . . . 10
- 2.4 Locality Sensitive Hashing . . . 12

3 Incremental Clone Detector 15
- 3.1 Overview . . . 15
- 3.2 Source Code Preprocessing . . . 18
- 3.3 Source Code Representation . . . 19
- 3.4 Clone Detection . . . 20
- 3.5 Output . . . 21

4 Incremental Clone Detection with Locality Sensitive Hashing 23
- 4.1 Motivation . . . 23
- 4.2 Overview . . . 24
- 4.3 Approach Decomposition . . . 26

5 Experimental Design **29**
5.1 Types of Experiments 29
5.2 Output Validation 31
5.3 Tools Configuration 31
5.4 The Infrastructure 32
5.5 The Corpus 32
5.6 Simulation of Commits 34

6 Experimental Results **35**
6.1 LIICD Measurements 35
6.2 LIICD vs SIG Measurements 40
6.3 LSH-based Extension Measurements 42

7 Discussion **49**
7.1 Main Findings 49
7.2 Implications 51
7.3 Threats to Validity 52
7.4 Applicability within SIG 52

8 Conclusion & Future Work **55**
8.1 Conclusion 55
8.2 Future Work 56

Bibliography **57**

A Corpus Collection **63**
A.1 Project Snapshots 63

B Excluded Directories & Files **65**

List of Figures

2.1 Locality Sensitive Hashing sub-operations 13

3.1 Substeps of the Clone Index creation workflow 16
3.2 Substeps of the Incremental Step workflow 17
3.3 Source code normalization step applied in Hummel's study [24] 18
3.4 Basic preprocessing steps applied in our study 19
3.5 Sliding window hashing based on CHUNK_SIZE 19

4.1 Substeps of the LSH-based Index Creation workflow 25
4.2 Substeps of the LSH-based Incremental Step workflow 26
4.3 Conversion of a pre-processed file into a set of shingles 26
4.4 Signature generation via Min Hashing 27

6.1 LIICD - Execution Time for the Index Creation flow 36
6.2 LIICD - Cumulative Memory Requirements 37
6.3 LIICD - Average Execution Time for the Incremental Step flow 38
6.4 Clone Detection for Ripple 39
6.5 Clone Detection for Kooboo 39
6.6 Clone Detection for Tensorflow 40
6.7 Clone Detection for OpenJDK 40
6.8 Cumulative comparison of SIG's detector versus the LIICD detector 41
6.9 Execution Time for the Index Creation flow for the two implementations 43
6.10 Memory requirements for the two implementations 44
6.11 Execution Time for the Incremental Step flow for the two implementations 45
6.12 Index Creation sub-measurements for the LSH-based implementation 46
6.13 Execution Time implications for varying number of hash functions during Min-Hashing 47

Chapter 1

Introduction

Technical Debt (TD) is a metaphorical term introduced in 1992 to explain the long-term implications of quick, hasty technical compromises on the health of a software system [11][36]. It refers to the trade-off between writing short-term, fast and messy code at the cost of increased maintenance efforts compared with writing long-term, easier maintainable, clean code, backed by deliberate thinking [7].

A form of Technical Debt is what is known as code duplication, a term that describes the duplication of source code in a software system [18]. In most cases, cloning occurs quite frequently due to the ease of copying and pasting code fragments by developers, when the business logic of a feature is similar or identical to existing code [45][51]. In fact, past studies report a significant amount of duplication found in examined software systems, ranging from 7% up to 29% in large codebases [29][47]. As a result, the maintainability of the underlying software project is negatively affected.

The existence of significant code duplication in a system's source code introduces a number of different problems. In particular, duplicates lead to increased codebase size and consequently to higher maintenance costs [4]. Furthermore, in case a bug is detected in one of the clone instances, every other instance has to be checked for the same bug and potentially be fixed [37]. The latter does not only necessitates one to be aware of a clone list but also requires an essential amount of time going through all the instances. Lastly, duplication can be problematic in terms of code comprehension as well as in future refactoring [4].

Automatically detecting the code clones in today's software projects is important since it sets the ground for future manual or automatic refactoring, leading to cleaner and more maintainable code. In that respect, various clone detection techniques have been proposed, the vast majority of which operate on the entire codebase of the underlying system. For each source code revision, such techniques use the whole system as input, regardless of the magnitude of the introduced changes. Although this might work well for stable legacy systems that are rarely updated, it is far from ideal in today's era of agile software development. When the clones for the next revision need to be detected, redundant calculations take place, adding up to the overall execution time of the detection process.

These drawbacks, along with the evolution of software development practices and the appearance of concepts such as continuous integration/development (CI/CD), agile, and

sprints, created the need for incremental approaches. In this context, the main idea is the reuse of information gained from the analysis of a revision to the next one, avoiding unnecessary time-inefficient operations.

This study is conducted in collaboration with the Software Improvement Group (SIG), a consultancy firm that focuses on software quality related challenges. One of the core focuses of SIG is related to the maintainability of software projects. More specifically, the company has developed automated tools that assess maintainability based on a number of predefined criteria, such as the volume and complexity of the code, the coupling of modules etc. One of these criteria is also the proportion of duplicated code within a project's codebase, for the detection of which a clone detector is used. That considered, SIG is highly related to the topic of this thesis and ideal to guide the development of a clone detector with real word applicability.

In this study, we develop a language independent, incremental clone detector and evaluate its performance within the context of SIG. SIG measures and scores code quality according to their maintainability model [22], measuring code duplication as one of their code properties. In our work, we refer to the developed technique as LIICD (Language-Independent Incremental Clone Detector).[1] We evaluate and compare our approach with the traditional clone detection approach that SIG uses, employing different software projects with varying sizes as input. Our experiments indicate improvements to the time needed to detect the duplicated code fragments. Moreover, we examine if Locality Sensitive Hashing (LSH) [25]—a nearest-neighbor similarity estimation technique—can be utilized to further extend and improve the LIICD detector, which we originally propose. Our first findings indicate that the LSH-based approach does not match the original approach's performance, although there is room for future research to explore possible refinements.

1.1 Problem Statement

Code duplication highly affects the maintainability of a software system [22]. Traditional techniques use the whole system as input, resulting in redundant calculations when the analysis has to be repeated frequently through time. For each code change, which in practice we could associate with a commit, traditional approaches need to re-run the clone detection process, processing the entire software system from scratch, regardless of the magnitude of these changes, i.e. regardless whether a single line in a single file was updated or multiple lines in multiple files were updated. Since searching for clones in a large code base can be expensive both in time and memory demands, the need for more fine-grained, commit-level approaches emerges.

The idea behind incremental techniques is the reuse of information across revisions so as to reduce unnecessary operations and bring down the time needed to detect the clones. Although a number of different incremental approaches have been proposed in the past, the vast majority of these are language-dependent, requiring a language parser to process the underlying code. On the one hand, this allows for a deeper analysis and the detection of more complex higher types of clones, but on the other, it comes with limitations when it

[1] Pronounced lai-cee-dee

comes to projects written in unpopular languages, for which finding or building a parser is challenging. Furthermore, if taken in the context of SIG, a language-independent clone detection approach allows to detect code clones uniformly across different programming languages. This, in practice, translates in maintainability findings that are language-agnostic. To our knowledge, an incremental technique that is purely language-agnostic is yet to be proposed.

In this study, we explore how such a language-agnostic incremental clone detector can be designed and developed. To do so, we build upon an existing text-based incremental approach, originally proposed by *Hummel et al.* [24]. Although the approach is not language-independent by default, we modify it to achieve the desired outcome. To observe the benefits that such an incremental technique can offer, we evaluate the detector in the context of SIG, comparing it with the company's existing detector. Furthermore, we examine LSH as a way to extend this approach, attempting to overcome some of the drawbacks of the LIICD detector. We then compare the two approaches with each other and investigate the potential of LSH in our context.

1.2 Research Questions

Current research in incremental code duplication detection focuses on language-dependent approaches trying to optimize and detect more complicated types of clones. Additionally, although a number of different studies, utilizing traditional techniques, touch upon language-independent clone detection, no studies have looked into how the same feature can be achieved in the context of an incremental technique. Thus, the research questions investigated in this thesis are the following:

1. **RQ1:** *How can we build a language-agnostic incremental clone detector and what kind of information do we need to store across revisions?*

   To answer the first research question we initially look into how traditional text-based clone detection approaches work. We specifically focus more on text-based detectors since this is the only category of approaches that does not require a parser, thus is suitable for a language independent approach. Apart from that, we try to learn from existing incremental approaches, studying the different stages of the clone detection pipeline and the intermediate information used between revisions.

2. **RQ2:** *How does the resulting LIICD (Language-Independent Incremental Clone Detector) approach perform?*

   We answer this question by looking into the performance of our clone detector in terms of two different evaluation metrics. In particular, we assess LIICD by measuring its time and memory requirements during the analysis of a number of different software systems, processing a series of commits for each system.

3. **RQ3:** *How does such an approach perform compared to the traditional, industrial-strength one that SIG is currently using?*

   SIG uses a broad range of software quality analysis algorithms embedded within its analyses tools. One of those is the clone detection logic responsible for the detection of code duplication and existing clones. To answer our question, we isolate the clone detection process of the respective tool's pipeline, measure its performance and compare it with the performance of our proposed method.

4. **RQ4:** *Can we use Locality Sensitive Hashing (LSH) to extend and improve the original approach?*

   To answer our last question we first look into how LSH can be used to extend our LIICD detector. Then, to investigate the potential of the developed LSH-based technique in the context of this study, we evaluate it using the same experimental setup as the one used to assess the performance of LIICD, and then compare the findings for two approaches.

1.3 Scope

The main focus of this study is to build a language independent, incremental clone detector, investigate its effectiveness and evaluate its performance. Detecting the clones is essential for the identification and removal of duplicated code and future maintainability of a software system. Knowing the existing clones, one can then proceed and manually or automatically refactor the code (e.g. by merging the duplicates in a single class or function) and as a result eliminate the drawbacks that come with it and make the codebase future-proof.

1.3.1 Languages & Techniques

Focusing on a specific programming language, especially a popular one such as Java [41], would allow for a deeper, more detailed clone detection. One such example is an approach capable of finding more complicated clones, such as semantically similar clones. In that case, it would also be reasonable to investigate different detection techniques such as token-based, tree-based, graph-based etc. However, SIG is not limited to the supported languages and finding or building a parser for less popular languages is challenging. In this context we focus on language independent detectors which can only be produced by leveraging text-based techniques.

1.3.2 Clone Types

There are four different types of clones, typically referred to as Type 1, Type 2, up to Type 4 [4][45][51]. In this study, we specifically put our focus on Type 1 clones, which refer to exactly identical fragments of code. This choice is based on the end goal of comparing our

study result with the existing state of the art approach used by SIG. Therefore, we design and develop our clone detector only accounting for Type 1 clones.

1.3.3 Clone Management

In the context of this study, we do not discuss or support clone management [13][48]. That means that we do not keep track of how the clones evolved throughout the different revisions of the underlying software project. On the contrary, we are interested in what happens in the context of a specific commit, meaning the clones that the specific code revision introduces or removes.

1.3.4 High Level Evaluation

Neither of our two developed incremental clone detectors is compared with existing state-of-the-art incremental techniques. Our aim in this study is not to discover whether these approaches perform adequately when compared with other relevant studies. On the contrary, our goal for LIICD is to evaluate its performance and examine it in comparison with SIG's current clone detection approach. Furthermore, in terms of our LSH-based approach, we investigate whether LSH can be used to achieve even better performance. Both our tools are publicly available,[2] thus additional studies examining these approaches from different perspectives can be conducted in the future.

1.4 Contributions

In the process of investigating the aforementioned questions, this study makes a number of different contributions. These are:

1. We investigate language independence in the context of incremental clone detection and develop LIICD, a tool based on an existing incremental text-based clone detection technique by *Hummel et al.* [24].

2. We investigate the suitability of LSH as a technique that can be leveraged to extend and improve the originally proposed approach.

3. We develop an incremental clone detector that adopts the LSH technique.

4. We evaluate LIICD's performance and compare it with SIG's commercial-grade clone detection approach to explore the benefits that an incremental approach can offer.

5. We evaluate our LSH-based in terms of efficiency, comparing it with LIICD and providing findings with respect to the performance of this extension.

[2] Both tools are available on Github: https://github.com/agamvrinos/LIICD

1.4.1 Incremental Detection

Our first goal in the context of this study is to develop and assess a language-agnostic incremental clone detection technique. In that respect, we first provide insights on the current status of the literature in the area of incremental clone detection and research relevant text-based techniques. Then, we develop a clone detector that satisfies our study's requirements, i.e. a detector that works incrementally and is capable of detecting Type-1 clones, while also being language-independent.

1.4.2 LSH Extension

In the context of our research goal, we further extend our incremental detector by utilizing Locality Sensitive Hashing (LSH), a technique used to efficiently search for near neighbors. In our case, this practically translates to efficient similarity estimation between source files. In particular, we compare the latter with the LIICD detector proposed, discuss the resulting findings and provide insights for further experimentation.

1.4.3 Evaluation

Having both the LIICD and LSH-based incremental approaches in place, we proceed to their evaluation. For our LIICD detector, we first measure its performance and then evaluate it within the context of SIG. More specifically, we isolate the clone detection process of SIG's SAT tool—a tool that SIG uses to analyze the quality of a software project and measure various metrics such as its maintainability—and compare it against our approach. For our LSH-based detector, we run experiments and evaluate it by comparing its performance with that of LIICD.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we describe the background of this study. This includes basic prerequisite knowledge on clone detection, as well as the related work in the particular scientific area. Then, in Chapter 3 we provide a detailed description of the proposed incremental clone detector. Chapter 4 introduces an extension to the original approach which adopts a widely known similarity estimation technique, known as Locality Sensitive Hashing. Both the original approach and the extension are then used to run our experiments, the setup and considerations of which are outlined in Chapter 5. Then, in Chapter 6 we present the results, followed by a discussion in regards to our findings in Chapter 7. Lastly, in Chapter 8 we summarize the findings and contributions of this study and provide suggestions for future work.

Chapter 2

Background & Related Work

In this chapter we discuss the background information required for this study. Furthermore, we dive into the existing literature and take a look into the scientific landscape within the area of clone detection.

2.1 Clone Detection Basics

This section provides information about the different types of clones, as these are identified by the related literature. Moreover, we look into the categorization of clone detection techniques according to the source code representation they adopt. Lastly, we discuss popular evaluation metrics, commonly used to assess the performance of clone detection techniques.

2.1.1 Clone Types

Code clones can be found in different forms and are not all the same. In fact, previous work in the existing literature [4][45][51] has identified and classified clones into four main categories. The first three categories describe syntactically similar code fragments, whereas the last one looks into the semantic similarity of two such fragments. These categories are summarized as follows:

- **Type 1:** An exact copy of the original fragment, only allowing for differences in whitespaces, blanks and comments.
- **Type 2:** Syntactically identical clones with variations in identifiers, literals, types and comments.
- **Type 3:** Syntactically identical clones with further modifications such as added/removed statements and variations in identifiers, literals, types, layout, and comments.
- **Type 4:** Code fragments that are semantically similar are identified as Type 4 clones. For instance, the functionality of an ”if” statement can be implemented using a ”switch” statement.

An example for each of these types of clones can be seen in the listings 2.2 to 2.5. In these examples, the higher the clone type the harder it is to detect the corresponding clones.

```
int myFunc(int n) {
    int sum = 0; // cm1
    int a = 2;
    for (int i = 0; i <=n; i++){
        sum = sum + a;
    }
    return sum;
}
```

Listing 2.1: Original code fragment

```
int myFunc(int n) {
    int sum = 0; // cm1
    int a = 2; // cm2

    for (int i = 0; i <= n; i++){
        sum = sum + a;
    }
    return sum;
}
```

Listing 2.2: Type 1 clone

```
float myFunc(int n) {
    float summary = 0; // cm1
    int a = 2; // cm2

    for (int i = 0; i <= n; i++){
        summary = summary + a;
    }
    return summary;
}
```

Listing 2.3: Type 2 clone

```
int myFunc(int n) {
    int sum = 0; // cm1
    // deleted line
    for (int i = 0; i <= n; i++){
        sum = sum + 2;
        i++; // new line
    }
    return sum;
}
```

Listing 2.4: Type 3 clone

```
int myFunc(int n) {
    int sum = 0; // cm1
    int a = 2;
    int i = 0;
    while (i <= n) {
        sum = sum + a;
        i++;
    }
    return sum;
}
```

Listing 2.5: Type 4 clone

2.1.2 Core Techniques

The majority of the existing clone detection techniques approach the detection of duplicated code in a conventional way. Over the last few years however, more and more focus is being given to incremental techniques that approach this on a more fine-grained level and attempt to avoid redundant recalculations by passing information across revisions. Regardless of the approach, detection techniques can be classified into four core categories, based on the way they represent the source code. The technique that a clone detector adopts, affects it in a number of different ways. For instance, one technique might be more suitable to detect higher types of clones compared to another but its performance might be worse. It's a trade

off. To better understand this we can think of an analogy in the context of the automotive industry. Car manufacturers build all sorts of different cars (clone detectors), but the engine types (core techniques) are limited to three or four, each with potential for a unique tuning (variations of techniques). The four techniques used in the area of clone detection are the following:

1. **Textual techniques**: These techniques look at code fragments in terms of textual equality, without applying any sophisticated transformations on the original source code. As a result, they achieve great coverage of Type-1 clones and are capable of detecting clones no matter the underlying language [51]. On the negative side, they are unsuitable for the detection of more complicated, higher types of clones.

2. **Lexical techniques:** In this approach, a sequence of tokens is extracted from the source code with the use of a lexer in a process called lexical analysis. Then, instead of source code sequences, token sequences are compared with each other. These techniques are usually very efficient in the detection of clones with minor differences (Type-2) such as renames [47].

3. **Syntactic techniques:** These are further split into *tree-based* and *metric-based* techniques. In the former, a parser is used to create an Abstract Syntax Tree (AST) and then its sub-trees are compared. Due to the tree abstraction of the source code, this category of techniques allows for the detection of more complicated types of clones. In the case of metrics-based techniques, a number of metrics is computed for each code fragment, and then the resulting metric vectors are compared in order to detect the clones.

4. **Semantic techniques:** These are also further categorized into 2 sub-categories, *graph-based* and *hybrid* techniques. In general, they employ static program analysis to allow for a semantic, rather than syntactic, comparison of two code fragments. Graph-based construct a Program Dependency Graph (PDG) from the source code and detect clone by comparing the sub-graphs. Hybrid on the other hand, use a combination of the aforementioned techniques.

2.1.3 Evaluation Metrics

In the area of clone detection, the proposed clone detectors are usually compared on the basis of three distinct evaluation metrics, namely *recall*, *precision* and *scalability* [47][54]. Although in this study we specifically focus in the scalability of the developed tools, the remaining two are also briefly discussed. These are defined as follows:

- **Recall:** Refers to the proportion of code duplicates within a codebase that a clone detector is capable of detecting. This is usually challenging to measure since it requires a benchmark to compare with. The following formula calculates recall [46]:

$$Recall = \frac{No. of\, correctly\, detected\, clones}{No. of\, total\, detectable\, clones}$$

- **Precision:** Refers to the proportion of the clones detected by a clone detector, which are true clones and not false positives. In the context of clone detection, precision is given by [46]:

$$Precision = \frac{No.\,of\,correctly\,detected\,clones}{No.\,of\,total\,detected\,clones}$$

- **Scalability:** Refers to the time & space requirements of the underlying clone detector, thus its ability to scale to larger codebases.

2.2 Clone Detection Challenges

The design and development of a clone detector comes inherently with several challenges. One of these is the design of a clone detection tool in such a way that it is capable of detecting multiple types of clones, while also scoring high accuracy and recall when applied in practice [51]. Especially the detection of semantic, Type-4 clones is considered a rather challenging task, identified as an undecidable problem by the existing literature [4]. Furthermore, many tools struggle with scalability, meaning they scale poorly when it comes to the analysis of larger codebases [50]. Lastly, portability, i.e. the ability of a detector to easily detect clones in multiple programming languages, is yet another challenge that the creators of detection tools have to face [51].

2.3 Clone Detection Techniques

In this section, we examine the various traditional and incremental techniques that have been proposed throughout the years.

2.3.1 Traditional Techniques

Traditional detection techniques are those that use the entire codebase of a software project as input to detect the clones regardless of the size of the changes introduced in a particular code revision.

Some of the most relevant text-based techniques are *Duploc* [15] and the techniques by *Marcus et al.* [39] and *Johnson et al.* [27]. *Duploc* uses a ”dot-matrix” to plot similar code fragments and then detects clones by parsing it while looking for patterns. *Johnson et al.* use substring matching to detect clones in a language-independent manner. Lastly, *Marcus et al.* use static analysis to analyze the codebase of a software system and determine semantically similar code fragments.

For the token-based techniques, *CCFinder* [28] is the most discussed approach in literature. In that, the authors use language-specific transformation rules to tokenize the raw source code. Then, a suffix-tree matching algorithm is used for the actual clone detection. *Dup* [2] and *CP-Miner* [37] have also attracted the community’s interest. The former utilizes a special kind of data structure, called parametrized suffix tree, to be able to detect

Type-1 and Type-2 clones, whereas the latter uses frequent subsequent mining, a data mining technique, to efficiently identify copy-pasted source code. Interestingly, more recent studies such as *CCLearner* [34] and *White et al.* [56] explore tokenization in combination with Deep Learning.

In the area of tree-based and metric-based techniques, *CloneDr* [3], *Deckard* [26] and the work of *Mayrand et al.* [40] and *Kontogiannis et al.* [31] are among the most widely used techniques, respectively. For tree-based approaches, *CloneDr* generates an annotated parse tree via the use of a compiler generator. During the detection process, subtrees are compared using characterization metrics. The authors of *Deckard* suggest a technique that uses a combination of Abstract Syntax Trees (ASTs) and Locality Sensitive Hashing (LSH) to efficiently cluster similar characteristic vectors and consequently detect clones. For metric-based techniques, *Mayrand et al.* calculate different metrics from names, layouts, expressions etc. of functions. Then, clones are detected by identifying functions with similar values for the calculated metrics. *Kontogiannis et al.* propose an approach that uses two methods for the detection of clones. The first method numerically compares values extracted by utilizing popular metrics and categorize code fragments to begin-end blocks. The second method uses dynamic programming to compute and report begin-end blocks using minimum edit distance.

Notable techniques in the category of graph-based techniques are *Duplix* [32], the detector proposed by *Komondoor et al.* [30] and *GPLag* [38]. The first two are quite similar, both being capable of finding maximal similar PDG subgraphs with high precision and recall. In contrast, *GPLag* uses PDG mining for the purposes of plagiarism detection. Contrary to other plagiarism detection approaches, GPLag can also detect plagiarism code that has intentionally been disguised.

Lastly, some hybrid techniques can be found in the works of *Funaro's et al.* [19] and the studies of *Agrawal et al.* [1] and *Saini et al.* [49]. *Funaro's et al.* propose a hybrid technique combining ASTs to identify code clones and a text-based technique to eliminate false positives. *Agrawal et al.* combine token-based with text-based approaches, exploiting the benefits of each, in order to detect Type-1, Type-2 and Type-3 clones, respectively. Finally, *Saini et al.* [49], propose a detector named Oreo, a tool capable of detecting Type-1 up to Type-4 clones with high accuracy and recall, utilizing machine learning, information retrieval, and software metrics.

2.3.2 Incremental Techniques

The studies that propose incremental techniques are scarce compared to those that introduce full system analysis tools. The majority of these focus on token, tree or graph-based techniques, which require a lexer or parser to construct the necessary data structures based on the corresponding programming language(s) of a system. In fact, most of these do not intend to support language-independence to begin with and thus produce detectors that, on the one hand are capable of detecting more complex types of clones, but on the other, require language-specific configuration.

Göde and Koschke [21] first introduced an incremental-based clone detection technique.

The authors proposed a tree-based technique, employing a global Generalized Suffix Tree (GST) data structure to represent the underlying source code. A tree-based technique was also used in *ClemanX* [42], in which each source file is represented as an AST. Each subtree of such an AST is further represented as a characteristic vector to allow for similarity comparison. Later, *Higo et al.* [23] suggested the first PDG-based incremental clone detection technique. In that, during the analysis phase, a PDG is constructed for each method of each updated file and stored in a database. Then, in the detection phase that follows, the user's manual input triggers the fetching of the appropriate PDGs from the database which are used to detect the clones. *Hummel et al.* [24] suggests an index-based algorithm. The core data structure used is a Clone-index, a global data structure resembling a typical inverted index. To our knowledge, although this technique uses a lexer to transform source code into tokens, it is the closest the literature has got to a language-independent incremental technique. Lastly, the recent *Siamese* of *Ragkhitwetsagul et al.* [44] employs a technique based on tokens that further produces four different code representations in order to be able to detect different types of clones.

2.4 Locality Sensitive Hashing

In principle, the procedure of finding near neighbors, for example similar documents, can be quite straightforward. Each document is compared with every other document based on a similarity metric. However, although the brute force approach works well for a small document pool, it scales poorly due to the time requirements that increase quadratically as the number of documents grows [5]. Approximate search schemes, such as Locality Sensitive Hashing (LSH), allow for a significant reduction in the computational time needed for this process [25]. In this section, we provide a high-level overview of the main concepts behind LSH. Furthermore, we briefly discuss various applications of it within the area of clone detection.

2.4.1 Definition

The main idea behind LSH is to hash the underlying data points a number of times using different hash functions, so as to ensure that similar items have a higher chance to collide and end up in the same hash bucket, compared to dissimilar items [33]. Then only the items that end up in the same bucket—also known as *candidate pairs*—are checked for similarity.

2.4.2 LSH Internals

There is a variety of different LSH schemes that can be used. The selection is determined by the underlying similarity measure that is used to compare the documents for similarity. In that respect, different measures such as the *hamming distance* [20], the *cosine distance* [53] and the *Jaccard coefficient* [6] can be used. In the case of cosine distance for example, the LSH scheme that is used is known as *SimHashing* [9] whereas in the case of Jaccard coefficient, the same is achieved via the use of *MinHashing* [10]. In our case, we put our focus on the Jaccard coefficient and Min-hashing since this approach was found to

outperform Sim-Hashing in a series of experiments [52]. Additionally, Min-Hashing is capable of detecting quite distant similarities (low percentages), something not possible with Sim-Hashing. Overall, the underlying components of this process are summarized in Figure 2.1.

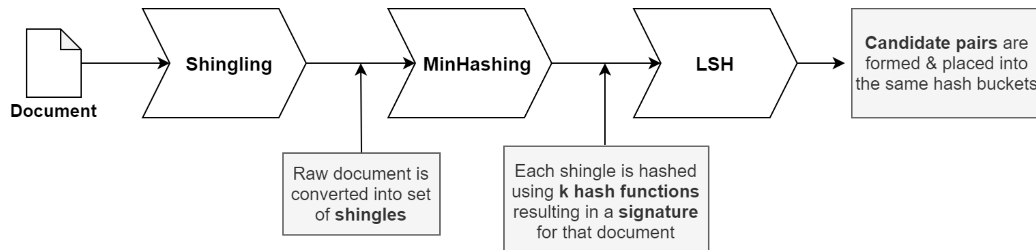

Figure 2.1: Locality Sensitive Hashing sub-operations

Figure 2.1 shows the particular LSH scheme consisting of three main sub-operations, *Shingling*, *MinHashing* and the actual *LSH*. These are defined as follows:

Shingling: This is the process of converting a document into a set of *k-shingles*. The shingles can be anything from simple *k-length* substrings, to combinations of *k* words. The choice of what we define as a shingle affects the probability of finding matches. For instance, for the extreme case of 1-character shingles, it will be much more likely to find a match in another document compared to the probability of finding a match for shingles of greater length.

MinHashing: The resulting sets of the previous step can be quite large, making their comparison rather inefficient. To partially solve this problem, MinHashing converts a set of shingles into a smaller fixed-length representation called *signature*. Then, rather than applying the Jaccard coefficient to the set of shingles, we instead apply it to the elements of the generated signature. That implies that the signatures need to be generated in such a way that the information of a set is preserved as much as possible. The application of this step though, does lead to loss of information, meaning that it is not anymore possible to calculate the exact similarity between two files. In contrast, the similarity is now calculated on the basis of an estimation, which can still lead to accurate results. All things considered, to generate the signature that represents a document, we hash every shingle of a set using *k-hash functions* and select the *minimum hash value* for each of these hash functions. For example, if we have 50 random hash functions, we'll get a MinHash signature with 50 values. As can be understood, the fewer hash functions we use, the higher the chance for collisions and thus the greater the error rate.

Locality Sensitive Hashing: MinHashing addresses the "curse of dimensionality" that comes with the sets of shingles. However, we still need to compare every signature with every other signature, which keeps the whole process inefficient. That is where the last step of the pipeline, LSH, finds its application. The technique used for this step is called *banding* where the matrix that holds the k hash values for each document, is split into *b bands*

each consisting of *r rows*. For instance, if we use 12 hash functions then we can split the matrix into 4 bands of 3 rows. Then, when a new document D_{new} comes in and we want to identify whether this document should form a candidate pair with another document, we go over every band and look for documents that share the exact same MinHash values on every row of that band. If we find a band for which all rows match, only then we proceed to execute the full comparison between these documents. Lastly, it should be noted that the selection of the number of bands and rows to be used is crucial since it determines the similarity threshold, over which we consider two documents similar.

2.4.3 Applications

Locality Sensitive Hashing has found applications in many different fields. In particular, *Verroios et al.* [55] and *Ebraheem et al.* [16] use it in the context of Entity Resolution. Additionally, the studies of *Cappeli et al.* [8] and *Zhou et al.* [58] adopt LSH-based techniques for biometric Fingerprint Indexing. Furthermore, LSH is quite frequently used in the context of recommendation algorithms and Collaborative Filtering. Google for example, uses LSH to personalize the widely known Google News [12], whereas *Zhang et al.* [57] use it to address the scalability issue of existing collaborative filtering algorithms.

In the context of clone detection, LSH has previously been used in a number of different past studies. More specifically, the authors of *Deckard* [26] propose a tree-based detector that uses LSH to compare the similarity of characteristic vectors extracted by the subtrees of the generated AST. *ClemanX* [42] employs a quite similar methodology, again for a tree-based approach. Lastly, *Hummel et al.* [24] suggests LSH as a future extension of their proposed detector to allow for the detection of Type-3 clones.

Chapter 3

Incremental Clone Detector

Our inspiration for the proposed LIICD approach derives from an existing study conducted by *Hummel et al.* [24]. Unlike the vast majority of incremental studies, which are built on top of tree-based and graph-based methods and are consequently strictly language-specific, the specific study introduces a solution based on text. As a result, this technique makes for a great base for experimentation towards a language-independent solution. Although the technique is not language-agnostic per se, due to the choice of the authors to include a tokenization step, we will see how minor adjustments can result in the desired outcome. More specifically, this chapter includes a discussion of the changes that were applied to achieve language independence and how these affect the resulting LIICD detector. Furthermore, we provide a detailed analysis of the detector's internals and explain how the various steps of the detection pipeline work.[1]

3.1 Overview

The incremental technique discussed in this study consists of two main workflows, each of which further includes a number of sub-steps. The first flow is related to the creation of an intermediate representation that is persisted and reused across a project's revisions. In the context of this work, this pool of intermediate information is known as the *Clone Index*. This flow uses the entire software project as input and runs only once at the beginning of the whole clone detection pipeline. Then, a second flow referring to the logic that runs the actual code duplication detection procedure outputs the discovered clones. This process is triggered when the underlying codebase has been updated, which in a real world setup would be when a new commit has been pushed to a version control repository.

3.1.1 Clone Index Creation Workflow

The workflow through which the clone detector creates the *Clone Index* contains a series of different steps. The whole process is depicted in detail in Figure 3.1. Initially, each of the files of the software project analyzed is fed into a preprocessing step which modifies

[1] The source code is available on Github: https://github.com/agamvrinos/LIICD

the code (e.g. by removing redundant blank lines) and determines the comparison granularity. During the next phase, the statements of the modified source code are grouped into sequences based on a predefined configuration parameter that determines the size of the group, and are subsequently hashed. Lastly, these hashes along with additional metadata such as the filename and index of a statement within a file are stored in the *Clone Index*.

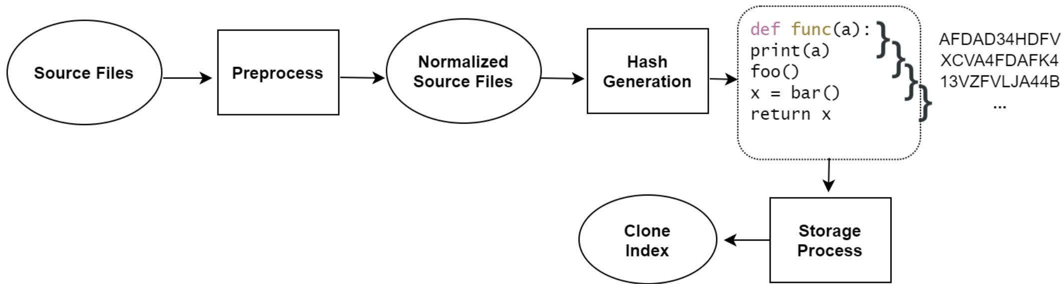

Figure 3.1: Substeps of the Clone Index creation workflow

Note that during this flow, we do not execute any clone detection logic to identify the initial state of the codebase as far as duplication is concerned. Thus, we initially do not have an overview of all existing clones in a system's codebase. Nevertheless, this is something that could be achieved by querying the *Clone Index* with each file of the initial codebase. However, since in this study we do not look into how the clones evolved, but are only interested in the clones that were removed or created in each new revision, we do not apply such a step.

3.1.2 Incremental Step Workflow

As explained earlier, the incremental step workflow is initiated every time a new commit is pushed to the version control-based repository hosting the related software system. In the context of this study though, due to the fact that the proposed detector is not integrated with an existing version control platform (e.g. in the form of a plugin), we manually simulate the procedure that would otherwise be automatically triggered in a real world setup. This is done via a JSON configuration file, which indicates the updated files along with the type of the corresponding update for every commit we are interested in analyzing. Such a configuration file can easily be generated from the information included in a typical commit. An example of such a file is illustrated in Listing 3.1, which shows a configuration file for the analysis of two commits, each with files that were either modified (M), added (A), deleted (D) or renamed (R).

```
{
  "commits": [
  {
    "id": "cb8f645e0f",
    "changes": [
      { "type": "M", "filename": "lib/plugins/loader.py" }
    ]
  },
  {
    "id": "564907d8ac",
    "changes": [
      { "type": "A", "filename": "fragments/test_refactor.yml" },
      { "type": "D", "filename": "fragments/arch_linux.json" },
      { "type": "R", "filename": [
        "test/facts/system/distribution/__init__.py",
        "test/facts/system/__init__.py" ]
      }
    ]
  }]
}
```

Listing 3.1: Example of a JSON configuration file indicating the changed files per commit

For the most part, the substeps of the incremental step workflow resemble those of the index creation workflow. In fact, the preprocessing and hash generation phases are identical, although in this case the related operations are only applied to the affected files, rather than the entire codebase. In the next step, the generated hashes are compared with those stored in the persisted *Clone Index*, during the process shown as ”Clone Detection” in Figure 3.2. A match of two hashes indicates the discovery of a clone. Nevertheless, as these hash values are generated by hashing a *fixed number of code statements*, say *N*, the discovered clones are at that point of length N. As a consequence, additional logic that expands the discovery so as to extend the clones to their maximal length, is also necessary.

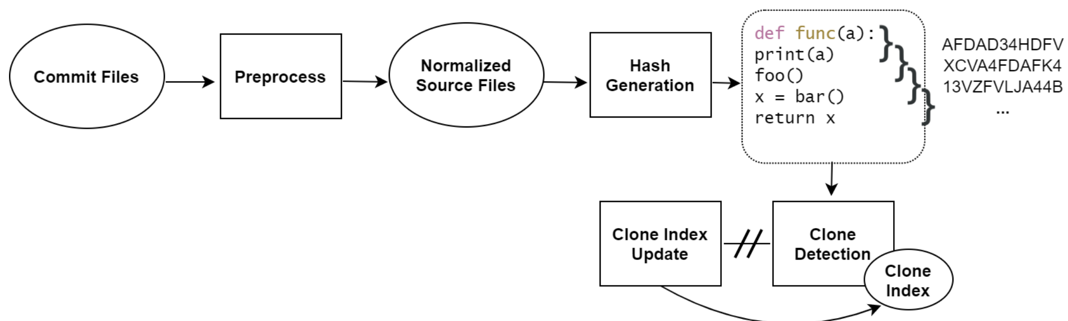

Figure 3.2: Substeps of the Incremental Step workflow

Lastly, as shown in the last steps of Figure 3.2, the *Clone Index* needs to be updated, so that the next iteration can compare the hashes generated by the files in a future commit with the ones persisted in the updated *Clone Index*. This process takes place simultaneously with the detection of the clones.

3.2 Source Code Preprocessing

In the approach of *Hummel et al* [24], the authors apply a normalization step that includes tokenization during the preprocessing phase of the source code. The purpose of this step is to convert the sequences of code into sequences of tokens allowing thus for the detection of Type-2 clones. An illustration of this, included in the original study, can be seen in Figure 3.3.

```
if (matching != null)
   matching.clear();
if (n1.isEmpty() || n2.isEmpty())
   return 0;
init(n1, n2, weightProvider);
prepareInternalArrays();
for (int i = 0; i < size1; ++i)
  augmentFrom(i);
// . . .
```

```
if(id0!=null)
id0.id1();
if(id0.id1() || id2.id1())
return int;
id0(id1, id2, id3);
id0();
for(id0 id1=int;id1<id2;++id1)
id0(id1);
// . . .
```

Figure 3.3: Source code normalization step applied in Hummel's study [24]

This step however, requires a parser so that source code elements such as identifiers, literals etc. can be converted into tokens. Unfortunately, this makes the whole process language specific since different programming languages require different parsers. Especially for less popular languages, finding or creating a parser can be quite challenging. That said, even if we wanted to maintain a collection of parsers and use the appropriate one based on the project's underlying language(s), that would still not meet our research goals.

Based on these facts, in our study we do not tokenize the raw source code but instead only apply basic preprocessing steps that do not affect the language-independence feature of the proposed detector. By doing so, we eliminate the possibility to detect Type-2 clones. However, in the context of this study, our main focus is on exact clones and therefore, a modification like that still allows us to meet the requirements set earlier in this study. Having said that, the preprocessing substeps we apply on the parsed source code are:

1. Removal of leading and trailing whitespaces
2. Removal of double whitespaces
3. Removal of blank lines

It should be noted that, in our case we do not remove comments since that would require additional work to identify all the different commenting styles corresponding to different

programming languages. Although this process does not necessarily require a parser, one would still need to segment the system into components of different languages in order to automate the process of the identification and removal of comments. A recent study by *Pascarella et al.* [43], which looks into source code commenting of open-source and industrial Java-based software systems, reports low percentages of code-to-comment ratio in these. In particular, for open-source systems, the measurements range from 6.3% to 12.1%, whereas for the industrial ones the numbers are smaller, ranging from 0.1% to 2.5%. That said, we do not expect large variations in our findings in the case comments were to be removed. As a result, we stick to the simple preprocessing steps mentioned above. An example of an application of these on actual source code, can be seen in Figure 3.4.

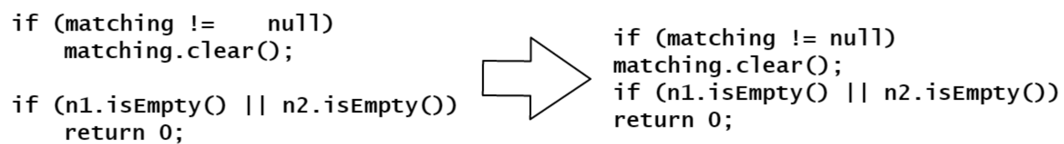

Figure 3.4: Basic preprocessing steps applied in our study

3.3 Source Code Representation

The proposed clone detector is text-based implying that there are no special transformations applied to the raw source code. However, the intermediate information persisted and reused by each code revision is not comprised of the simple preprocessed statements. Instead, blocks consisting of a fixed number of statements are hashed and the resulting hash values along with metadata is the actual information stored in the *Clone Index*. This happens on *sliding window* basis where consecutive blocks of size *CHUNK_SIZE* are hashed one after the other starting from the range [0, CHUNK_SIZE - 1] all the way up to [LINES_COUNT - CHUNK_SIZE, LINES_COUNT - 1]. An example of this process with a predefined CHUNK_SIZE set to 2 can be seen in Figure 3.5. The choice of a value for the CHUNK_SIZE inevitably also determines the minimum length of a clone.

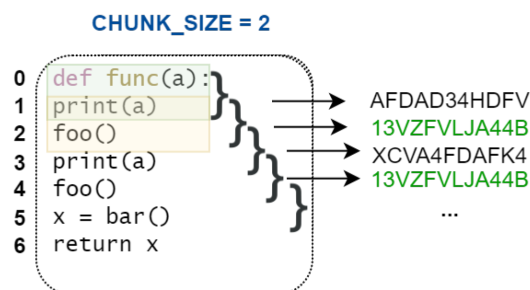

Figure 3.5: Sliding window hashing based on CHUNK_SIZE

Every hash value for each hashed block is then stored in the *Clone Index* along with additional meta information, used during the phase of the actual clone detection. That is:

1. the *filename* of the file that contains the hashed block
2. the *statement index*
3. the *start line* of the block
4. the *end line* of the block

Lastly, it should be noted that the *Clone Index* can either be persisted in memory or in an actual database. In our case, we persist this information in memory, although in a real-world setup an actual database would be necessary.

3.4 Clone Detection

The clone detection process is initiated for each new code change, or commit. The detector preprocesses each file included in that commit and proceeds by hashing the consecutive blocks of code based on the predefined CHUNK_SIZE. The generated hashes are then compared with those stored in the *Clone Index*. Any pair of matching hash values indicates the discovery of a clone that consists of exactly CHUNK_SIZE lines. However, this alone is not enough since a clone might consist of more lines than those determined by CHUNK_SIZE. Therefore, the detector runs additional logic that investigates whether the minimal clone that was identified, can further be extended. The details of this process are analytically explained in the study by *Hummel et al.* [24]. However, on a high level, this works by detecting whether the individual clone instances overlap. For instance, if a clone comprises 5 lines of a file starting from index 0 up to index 4 and another clone in that file is found between the lines with index 1 and 5, that automatically means that these blocks overlap and the individual clones can be merged into a single one occupying lines 0 to 5. Note that that this merging process is only possible because of the nature of this detector, which specifically focuses on identical blocks of code. This assumption would not be valid in case of fuzzy clone detection based on estimations.

During the detection of the clones, the clone index is also updated and prepared for the next time the incremental step flow gets triggered when a new commit arrives. More specifically, the file changes included in a commit are handled in batches based on the type of the change. This happens as follows:

- We first handle the batch corresponding to deletions. We query the Clone Index with the index entries of the deleted files to identify which clones are removed, and then remove these entries from the index. Handling the deletions first is important, because otherwise (e.g. if we handled the updated files batch first), we would compare the respective index entries with an outdated Clone Index.

- We then handle file updates corresponding to renames. This case is similar to the handling of the deletions. We detect the clones and remove the entries corresponding to the old filenames and then again add them add new entries matching them with the updated filenames.

- The next batch in order is that of the updated files. We treat this case as deletions followed by creations. That means that, as in the case of deletions, we first query the index with the non-updated versions of the files to find which clones are removed and then remove the outdated entries from the index. Subsequently, we generate the index entries for the updated versions of the files and query the index to find the clones that were added. Finally, we update the index with the new entries.

- Lastly, for the simple case of newly created files, we generate the corresponding index entries, detect clones and eventually update the index by appending the entries to the Clone Index.

3.5 Output

The output of the detector is in the form of raw text logs indicating the clones that were discovered. More specifically, these logs include: (1) the *filename* for each clone instance, (2) the *starting index* of each clone in the preprocessed file, (3) the *start and end lines*, and (4) the number of blocks of code of CHUNK_SIZE length that contributed to the final maximal clone. An example of this can be seen in Listing 3.2. The specific output corresponds to the files shown in Listing 3.3 and Listing 3.4.

```
1 (.../project/Test.java|0|0-6) -
2 (.../project/Test2.java|1|1-7) - 2
```

Listing 3.2: Example of the detector's output

```
0 import java.lang.*;
1 import java.io.*;
2 class Test {
3     public static void main(
      String []args) {
4         System.out.println("Hello
       world");
5     }
6 }
```

Listing 3.3: Test.java

```
0 import java.util.*;
1 import java.lang.*;
2 import java.io.*;
3 class Test {
4     public static void main(
      String []args) {
5         System.out.println("Hello
       world");
6     }
7 }
```

Listing 3.4: Test2.java

In this case, the clone detector detects a clone between files *Test.java* and *Test2.java*. In the former, the clone instance can be found at lines 0 to 6, whereas in the latter at lines 1 to 7. Note that the indices correspond to lines after a file has been pre-processed. That means that for example, if Test.java had a blank line at index 0—thus the clone was moved to lines 1 to

7 of the original file—, the clone instance would still be detected to be at lines 0 to 6 because of the removal of the blank line during the preprocessing phase. Finally, the detector also outputs the number of blocks of code that contributed to the resulting clone. In this example, the selected CHUNK_SIZE was chosen to be 6. As a result, the number 2 indicates that two overlapping blocks of code of length 6 were used, resulting in the detection of a clone of length 7.

Chapter 4

Incremental Clone Detection with Locality Sensitive Hashing

In Chapter 3 of this study we discussed how the original incremental clone detection algorithm proposed by *Hummel et al.* [24] works and how it can be refined to achieve language independence. However, a closer look at the empirical results reported in the original study raises concerns about its performance. In this chapter we introduce an extension of the incremental clone detector discussed in Chapter 3. For that purpose, we use the technique we introduced in section 2.4, known as Locality Sensitive Hashing (LSH) and investigate its suitability in the context of the LIICD detector. More specifically, we first explain the motivation about this extension. Then we dive deeper into this approach's internals and discuss how it differs from the original approach and which parts of it remain unaltered.[1]

4.1 Motivation

Our motivation to build an extension of the approach mentioned earlier emerges from the findings of the experiments conducted by the authors of the study of *Hummel et al.* [24]. More specifically, in one of the experiments the authors analyze the Eclipse SDK (v3.3), a software project consisting of more than 42 million lines of code and more than 200,000 Java source code files. In that, the authors measure the time needed to create the initial *Clone Index* and the time required to query and update it. The results of this experiment can be seen in Table 4.1.

Index Creation (complete)	7 hr 4 min
Index query (per file)	0.21 sec median 0.91 sec average
Index update (per file)	0.85 sec average

Table 4.1: Eclipse SDK (v3.3) analysis measurements [24]

[1] The source code is available on Github: https://github.com/agamvrinos/LIICD

Looking at these measurements, it is easy to see that the time needed to create the *Clone Index*, measured at 7 hours and 4 minutes, is rather extensive. Although the authors ran the specific experiment on rather old—for today's standards—hardware, this is still a substantial time requirement. Our goal with the proposed LSH-based extension of this chapter, is to investigate whether LSH can be utilized to improve the performance of this step and eventually reduce the time needed to create the intermediate information.

4.2 Overview

The main idea behind this approach is to avoid calculating the entire *Clone Index* from scratch, an operation which according to the original paper seems to be very time-expensive. To do so, we use LSH to efficiently estimate the similarity of the files of a software project. Then, for those files that were found to be similar, we calculate the index entries on the fly and perform the detection operation in the exact same way as the one discussed in Chapter 3. Of course, this approach comes with a trade-off. Although intuitively it looks like it could offer an improvement with respect to the detector's performance during the index creation flow, it negatively affects the following two aspects:

- **Recall:** With this approach, the clone detection process only runs for files that were found to be similar. Therefore, it is logical to assume that there will be clones that will be missed, for example in the case of large files that differ for the most part but have some identical blocks of code. To decrease the chance of missing clones, we can select a low similarity threshold when comparing the files, although eliminating this possibility altogether is not realistic.

- **Query Performance:** Calculating the index entries on the fly for the pairs of similar files adds extra calculation overhead and thus increases the time needed to query and detect the clones when a new commit occurs. However, we are tolerant to this behavior if that means that the creation time of the index drops significantly.

Similar to the clone detector introduced in Chapter 3, here as well, we can distinguish two main workflows. The first flow is again related to the creation of initial intermediate information to be reused across revisions. In this case however, this information is different than the *Clone Index* of the LIICD detector. On the other hand, the second flow is once again triggered for every new revision of a software system. Via the use of LSH, the affected files are placed in buckets along with already existing files which are similar to each of them above a predefined similarity threshold.

4.2.1 LSH-based Index Creation Workflow

Similarly to the Clone Index creation flow discussed in Chapter 3, this flow includes a number of substeps, eventually leading to the generation of the initial state of the intermediate information. The starting point of this pipeline is the same as in the LIICD detector, meaning that the exact same preprocessing steps are applied. However, the remainder of the flow

is significantly different. In particular, instead of hashing blocks of code and storing them in what we called a *Clone Index*, we apply LSH with the purpose of grouping similar files together. To do that, we employ the LSH substeps explained in section 2.4.2. These are, the shingling and generation of minhash signatures for the given files, as well as the grouping of similar ones. Each of these are described in detail in the following sections of this chapter. Overall, a high-level representation of this process can be seen in Figure 4.1.

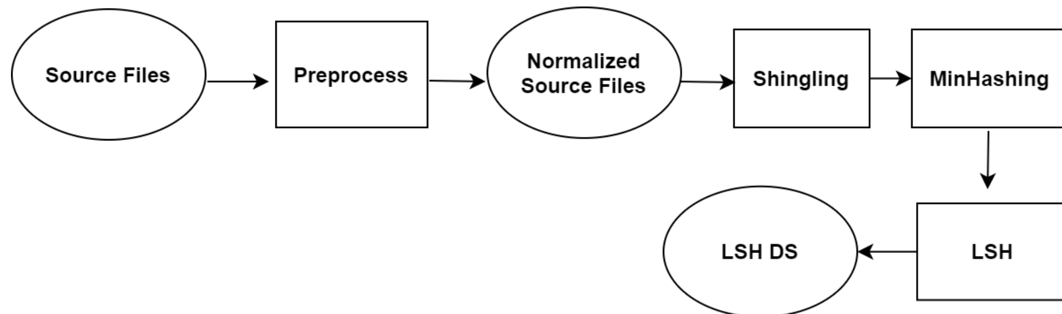

Figure 4.1: Substeps of the LSH-based Index Creation workflow

As in the case of the LIICD detector, we call the collection of persisted intermediate information, *Index* in what can be seen as LSH DS in the Figure 4.1.

4.2.2 LSH-based Incremental Step Workflow

Similar to the respective flow of our LIICD approach, the incremental step is initiated every time the codebase of the underlying software project gets updated to a new revision. Normally, the preprocessing operations for this flow remain the same. The rest of the pipeline however is different than the LIICD approach. More specifically, in the case of the LSH-based detector, we first identify the files in the existing codebase, with which each file affected by a commit is similar. To do so, we apply LSH and query the LSH-based *Index* which results in each file ending up in a hash bucket with similar files. For the similar files, we follow the same approach as in the LIICD detector, generating the hash values and performing the exact same clone detection process. A high level overview showing the substeps of this flow can be seen in Figure 4.2. It should be noted that in this case as well, the commits are represented via a JSON configuration file.

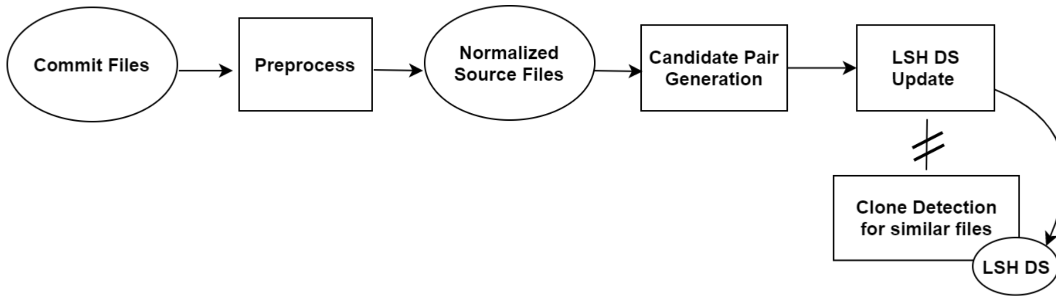

Figure 4.2: Substeps of the LSH-based Incremental Step workflow

4.3 Approach Decomposition

In this section we break down LSH in the context of our extension and discuss decisions concerning each individual substep of our implementation. For the purposes of this extension, we used *datasketch*,[2] a third party library that comes with an embedded MinHash LSH implementation.

4.3.1 Shingling

As explained earlier in section 2.4.2, *shingling* is the process of converting a raw document into a set of shingles. In our case, these documents are the source files of the project under analysis. After the preprocessing step takes place, we convert the resulting file into a set of shingles by taking each line of that file as a shingle. The outcome of this process can be seen in Figure 4.3, which illustrates how a preprocessed file is converted into a set of shingles corresponding to the lines of that file.

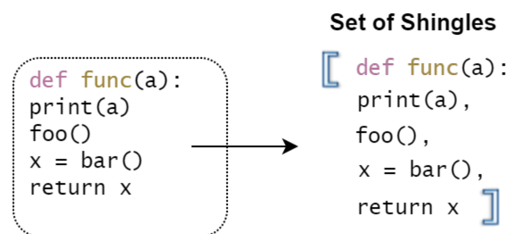

Figure 4.3: Conversion of a pre-processed file into a set of shingles

4.3.2 MinHashing

The next step in the overall LSH algorithm pipeline is *MinHashing*. As a reminder, the purpose of this step is to eliminate the phenomenon of large sets of shingles which drastically increase the time required to calculate our similarity metric, the Jaccard coefficient. This is

[2]The library is available on Github: https://github.com/ekzhu/datasketch

done via the use of *k-hash functions* which are used to hash every shingle within the set of shingles for each individual file. Then, the algorithm selects the lower hash value for each of the k hash functions and by doing so generates the signature. This second step of the pipeline is depicted in Figure 4.4.

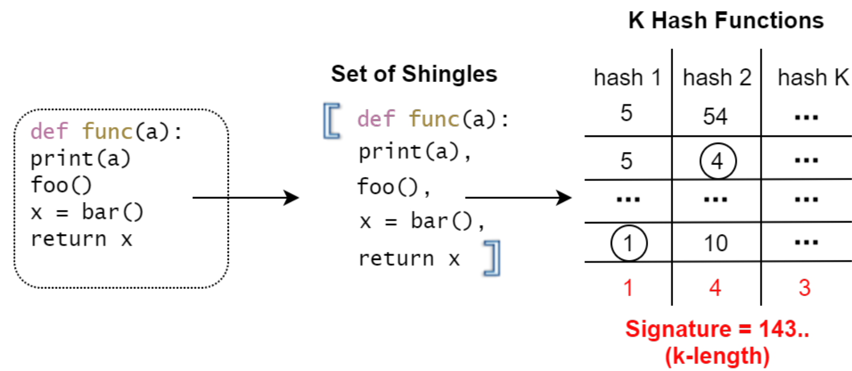

Figure 4.4: Signature generation via Min Hashing

This conversion comes with the cost of loss of information, since the sets are translated into fixed-length signatures, which can't be used to calculate exact similarities. However, it has been demonstrated that in this case we can still get quite accurate estimations [33]. More specifically, the accuracy of the estimation is a function of the number of hash functions used, thus the length of the signatures. The larger it is, the better the estimation but the more time-intensive the process of calculating it. In fact, the *error rate* of the similarity estimation is given by the formula:

$$error = 1/\sqrt{k}$$

where k represents the number of hash functions, meaning that for example, for 256 hash functions, we get a 6.25% chance of false negatives or false positives. Datasketch supports the alteration of the number of hash functions via the use of a configuration parameter.

4.3.3 Locality Sensitive Hashing

The final step of the LSH scheme deals with increased computational time needed to compare each signature with every other signature. As mentioned in Chapter 2, this is handled via a technique called *banding*. Here, the matrix that contains all the hash values for every shingle of a set, is split into b *bands* each consisting of r *rows*. Then, we consider a document D_1 to form a *candidate pair* with another document D_2, if every row of a specific band of D_1 matches with every row of the corresponding band in D_2. If that is the case, the documents are considered candidate pairs and end up in the same hash bucket. Although, in a custom LSH implementation we would have to pick the values for b and r ourselves —always paying attention to how these affect the similarity threshold—Datasketch works the other way around, allowing us to define the threshold we want, automatically calculating the proper values for these parameters.

Chapter 5

Experimental Design

In this chapter, we describe the quantitative experiments that we run and explain how these are aligned with the goals of this study. Furthermore, we mention the various evaluation metrics that we use and examine their suitability for each individual type of experiment. Lastly, we describe the process of our corpus collection and discuss how we simulate a series of incoming commits for the purposes of our analysis.

5.1 Types of Experiments

In this study, we design and run experiments with the goal of answering the research questions defined earlier in section 1.2. Out of these, our first question does not require running such experiments, since it is already answered by the fact that we were able to develop two clone detector implementations with the desired features to begin with. For the rest, we answer them by running quantitative experiments, corresponding to the research question we desire to answer in each case. These are described in detail in the following sections.

5.1.1 LIICD Evaluation

With respect to our second research question, we first seek to identify how our language-agnostic incremental clone detector, LIICD, resulting from adjusting Hummel's [24] approach, performs. To do so, we run LIICD for five software systems and measure its time and memory requirements to gain insights regarding its performance. More specifically, for each system, we analyze a series of 50 commits and measure the time and memory needed for the index creation flow and the average time needed to process the 50 commits during our incremental step flow. We run additional experiments to observe LIICD's behaviour with respect to the detected clones. In particular, we analyze the 10 most recent commits for each system in our dataset, and keep track of the clones that were added and removed.

5.1.2 LIICD vs SIG Evaluation

To answer our third research question, we run experiments with the purpose of comparing the performance of our LIICD approach with that of SIG's state-of-the-art traditional clone

detection approach. The ultimate goal here is to verify the improvement that an incremental approach provides, when the detection process is regularly repeated for every new software project revision.

SIG's clone detector is different from LIICD in many aspects, making a direct comparison a challenging task. More specifically, SIG's clone detection process runs within SAT, a tool that runs a lot more additional processes other than clone detection. This alone, requires that we isolate the clone detection process and measure it separately from the rest of the processes. Furthermore, SIG's detector is not incremental and thus outputs all the clones in a software system's revision. In contrast, LIICD is designed to only output the newly added or removed clones in a specific system revision. Consequently, it becomes obvious that the output of the two approaches in terms of clone detection is not directly comparable.

In the context of this study, we measure and compare the two approaches on the basis of the elapsed time they require for the clone detection process. Specifically, we deploy SIG's detector for a series of 50 commits, measure the average time needed to process these and compare the findings with those of LIICD's individual evaluation. With respect to the clones each tool detects and considering the described difference in each approach's output, we do not verify in practice whether the detected clones for the two detectors match. However, given that both our approach and SIG's detector are non-probabilistic, text-based techniques—meaning that the raw source code does not undergo any transformation that could lead to loss of information, thus reduced precision and recall—, we can expect the detected clones of the two approaches to match. In terms of memory, the approaches are again not comparable due to the complexity to isolate SIG's clone detection process and specifically take memory measurements only for that part of the process.

5.1.3 LSH-based Extension Evaluation

Regarding our last question, we are interested in measuring the performance of our LSH-based extension and compare it with that of LIICD. Similar to LIICD's evaluation, we achieve this by measuring the execution time and memory needs of our LSH-based extension and compare these with the respective findings for the LIICD approach. Since in this case both approaches are incremental and are built out of the same core idea, we can do a one-on-one comparison by comparing our metrics for each of the two flows, the index creation flow and the incremental step flow. Again, we use the same experimental setup, using five software systems and 50 commits for each of these.

Note that in the context of this study, our research question only aims to examine the developed LSH-based extension in terms of efficiency, meaning the performance boost that this technique might offer in either of the two workflows of index creation and incremental step. Consequently, although it would also reasonable to also measure the recall for the LSH-based implementation—considering that this approach by definition involves a probability of missing clones—, we do not run such experiments. Although findings about clone loss would be informative, these only makes sense after first having identified whether an LSH-based implementation is efficient enough to improve the LIICD approach, to begin

with. As a result, in this study we only investigate the efficiency of the approach and leave related empirical experiments about clone loss as a future work.

5.2 Output Validation

The validation of the output of our developed detectors, meaning the assessment of whether the detected clones are valid and complete (i.e. all the available clones have been detected) requires a benchmark for the selected software systems. Such a validation dataset was not available to us during the development of this study. Due to this, we instead manually validated our LIICD and LSH-based approach's output, in the context of a small-scale project of ours. More specifically, we have tested and validated multiple use cases regarding the deletion, update, creation or renaming of a file and crafted commits manually to simulate the updates of an actual software system. Naturally, for the LSH-based detector we did not check for completeness due to the nature of this approach. Note that although we only tested the output for a small-scale project, we expect our approach to work identically for larger codebases as long as all use-case scenarios are covered. The tests that we ran refer to the following use cases:

1. *Addition of newly created files:* We tested scenarios where these files introduced a single or multiple clones and others where no duplication was added. In the former type of scenarios, we verified that the clones were correctly detected, whereas in the latter, we verified that no clones were output.
2. *Renaming of files:* With respect to renaming, we tested that the clones in renamed files were detected and logged with the updated filename.
3. *Update of files:* We tested multiple scenarios where we removed and added blocks of code in existing files, testing the detection of removed or added clones.
4. *Deletion of files:* We tested the deletion of files containing clones, in which case we verified that the removed clones were correctly detected.

5.3 Tools Configuration

Table 5.1 indicates the configuration parameters used for the LIICD and LSH-based clone detector implementations for the purposes of our experiments. The *CHUNK_SIZE*, which indicates the number of lines in each hashed block of code, hence the minimum clone size, was selected to be 6, indicating an identical minimum clone length as the one SIG is currently detecting by default. The *PERMUTATIONS* parameter of our LSH-based implementation, refers to the number of hash functions used for the process of MinHashing. A value of 64 results in an error rate of $error = 1/\sqrt{64} = 12.5\%$ in terms of similarity comparison. That means that when two files are identified as similar/not-similar, there is 12.5% chance that this identification is erroneous. Lastly, a *THRESHOLD* parameter of 0.2, translates to a percentage of 20%, indicating the lowest threshold of similarity for which two files are detected as similar.

LIICD	LSH-based		
CHUNK_SIZE	**CHUNK_SIZE**	**PERMUTATIONS**	**THRESHOLD**
6	6	64	0.2

Table 5.1: Configuration for the LIICD & LSH implementations

5.4 The Infrastructure

For the purposes of our experiments, we run the two categories of tests on a single machine with the hardware setup as shown in Table 5.2.

	Specifications
Memory	32 GB
CPU	Intel Xeon E5-2650 v2 @2.6GHz

Table 5.2: Hardware specifications

5.5 The Corpus

In the context of our experiments, we use five open source projects as input to the various implementations of detectors. These were chosen in such a way so that we can cover a relatively wide range of projects with varying sizes and programming languages. More specifically, we measure the size of each project in terms of LOCs (Lines Of Code), using *CLOC*,[1] a tool capable of counting blank lines, comment lines, and physical lines of source code in many programming languages. Due to the nature of our detectors, which output the clones added or removed in a specific revision instead of all the clones in that revision, measuring the exact proportion of duplication in the selected systems is not possible. However, statistics extracted from the analysis of 192 software systems, conducted within SIG, measured the average duplication to be approximately 13% (std: ±12%). In the rest of this section, we present the process of measuring the LOCs for each system as well as the process of filtering out nugatory parts.

5.5.1 Initial Corpus Collection

For the collection of the initial corpus, we measured a large number of open source projects and selected five of them, with their LOCs ranging from approximately 300,000 up to about 23,000,000 lines of code. The *CLOC* configuration parameters used during this phase can be seen in Listing 5.1.

[1] The tool is available on Github: https://github.com/AlDanial/cloc

```
1  $ cloc --skip-uniqueness {target_project}
```

Listing 5.1: CLOC Initial configuration

The selected projects can be seen in Table 5.3. The reason we did not select software systems with LOCs below the described range, is because the true benefits of an incremental approach only become visible for projects with a relatively large number of LOCs, when repetitive detection in traditional approaches becomes sluggish.

	Main Languages	**No. of Files**	**No. of LOCs**
Linux Kernel	C	57,205	23,229,768
Openjdk-14	Java	60,444	12,045,316
Tensorflow	C++, Python	12,387	3,194,893
Kooboo	C#, JS, HTML, CSS	4,109	670,265
Ripple	C / C++	1,399	312,011

Table 5.3: The initial corpus of open source projects

As can be seen, the selected pool of software projects covers a wide range of programming languages and LOCs. The presented values for the *No. of LOCs* column, result from the addition of the LOCs corresponding to comments and those referring to actual source code. We do not take into account the empty lines, since these are removed during the preprocessing phase of each detector.

5.5.2 Filtering Production Code

SAT, the tool that SIG uses to measure the various maintainability metrics, including clone detection, runs the corresponding analysis only on parts of the codebase that are marked as *production code*. In contrast, code marked as *test code* along with other insignificant files, such as the README.md or logging files, are ignored. As a result, SAT only processes part of the entire codebase, which logically leads to a decreased number of processed LOCs. For a comparison between SAT's detector and our implementation to be objective, we need to also account for this removal. Therefore, we again use CLOC to estimate the LOCs for each project, excluding this time specific irrelevant directories. The exact configuration can be seen in Listing 5.2. The new measurements for the aforementioned projects can be seen in Table 5.4.

```
1  $ cloc --skip-uniqueness --exclude-dir=test,tests,doc,examples,licences
     ,lib --not-match-f=^.*test.*$ {target_project}
```

Listing 5.2: CLOC Filtering configuration

	Main Languages	No. of Files	No. of LOCs
Linux Kernel	C	56,270	22,963,637
Openjdk-14	Java	23,905	7,498,482
Tensorflow	C++, Python	8,610	2,120,650
Kooboo	C#, JS, HTML, CSS	4,064	668,055
Ripple	C / C++	1,073	207,166

Table 5.4: The filtered corpus of open source projects

5.5.3 Filtering Invalid Files

Every software project, including those selected in our corpus, usually consists of a number of files in binary format. Processing such files in the context of clone detection is meaningless since there are no valuable insights that can be extracted. Consequently, for both the LIICD and LSH implementations, we further exclude binary files and files with non-unicode characters, when processing each corresponding software project. A list stating all the file extensions that were filtered out can be seen in Appendix B. Note that the removal of such files, is expected to only marginally affect the number of LOCs mentioned in the previous sections, as such files typically form only a very small proportion of the entire codebase.

5.6 Simulation of Commits

To observe and measure how the detectors perform during the incremental step workflow, we need to simulate the process of commits being pushed to the version control repository hosting the project under analysis. As explained in Chapter 3, this simulation happens via the use of JSON configuration files that consist of a list of the commits that we want to analyze, along with the changes (creations, updates, deletions, renames) that each of these commits introduces.

In the context of our experiments, we analyze the *50 most recent commits*—excluding merges—for each of the five software projects and measure the metrics corresponding to each of the described categories of experiments. The most recent commit for each software project, indicating the snapshot we used at the time of our analysis, can be found in Appendix A. Since some of these 50 commits might consist of file changes for files that our detectors by definition exclude, there is a chance of processing less than 50. For instance, if a commit only includes modifications of test files (which are skipped by the detectors), then this commit is skipped, because no files are eventually processed.

Chapter 6

Experimental Results

This chapter presents the results of the experiments discussed in Chapter 5. More specifically, we first take a closer look into the outcome of our experiments regarding the LIICD detector, introduced in Chapter 3. Then, we present the findings of our experimental attempts to measure the performance of SIG's traditional clone detection approach and compare it with the proposed incremental clone detector. Lastly, we provide the results of our experiments investigating the efficiency of our LSH-based extension and examine these comparatively along with the findings of our LIICD experiments.

6.1 LIICD Measurements

The exact measurements for the experiments regarding the evaluation of our LIICD detector are depicted in Table 6.1. As shown, the LOCs for each project are within a margin error of approximately 2.7% when compared to the original estimates extracted through CLOC, apart from Tensorflow and OpenJDK, for which the excluded directories and file extensions seem to comprise a significant number of the overall codebase. Furthermore, most commits out of the 50 that were analyzed for each codebase were processed. As a reminder, the skipped ones refer to commits affecting *only* files that are ignored by our implementation in the first place (e.g. test files). The next two columns of the table present the average elapsed execution time of the index creation and incremental step workflows for the LIICD implementation. Finally, the last column shows the standard deviation for the incremental step time measurements.

Project	LOCs Read	Commits Processed	Index Creation Time (sec)	Average Incremental Step Time (sec)	Incremental Step standard deviation (sec)
Rippled	208,100	42	3.75	0.85	1.31
Kooboo	681,143	50	16.28	0.03	0.04
Tensorflow	3,814,652	45	65.89	4.29	3.32
Openjdk-jdk14u	3,377,211	46	48.53	4.72	5.07
Linux Kernel	23,603,823	45	321.21	N/A	N/A

Table 6.1: LIICD measurements

6.1.1 Index Creation

Time Measurements

Figure 6.1 illustrates the time measurements for the index creation flow of our LIICD detector. As shown, the overall time needed by LIICD for the index creation ranges from a few seconds for smaller systems, such as Rippled, up to approximately five minutes for large ones, such as the Linux Kernel, implying that the elapsed time for the completion of this step is affected by the size of a software system. This is expected considering that systems with a greater number of source files and LOCs lead to larger clone indexes, requiring more time to calculate. Nevertheless, the overall index creation time fluctuates in rather low levels, given that this process took at most about five minutes for a system as large as the Linux Kernel, consisting of more than 20 million LOCs.

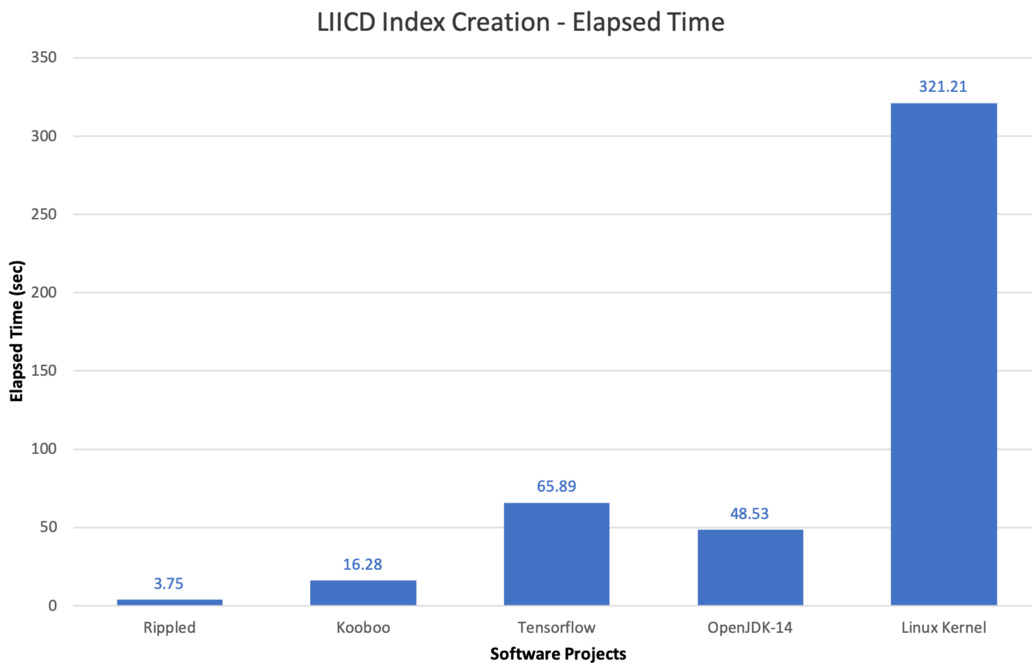

Figure 6.1: LIICD - Execution Time for the Index Creation flow

Memory Measurements

The memory needed by our LIICD approach to analyze the five software systems in our dataset can be seen in Table 6.2. Although these measurements were taken during the entire analysis of these systems, including both the index creation and incremental steps, the reported numbers mainly correspond to the memory requirements of the former. As explained in section 6.1.2, the incremental steps were, for the most part, observed to execute in a matter of a few seconds, leading to short spikes in the memory usage. Therefore,

their reporting is not of significant importance when considering the overall memory usage required by the specific detector implementation.

Project	Memory (MB)
Rippled	129
Kooboo	348
Tensorflow	1791
Openjdk-jdk14u	1712
Linux Kernel	12500

Table 6.2: LIICD - Memory measurements

Figure 6.2 better illustrates the memory our LIICD approach used during each analysis. As can be seen, the measurements for the first four systems are rather low, considering today's memory standards. However, for the considerably larger system of the Linux Kernel, the memory usage rises to much higher levels.

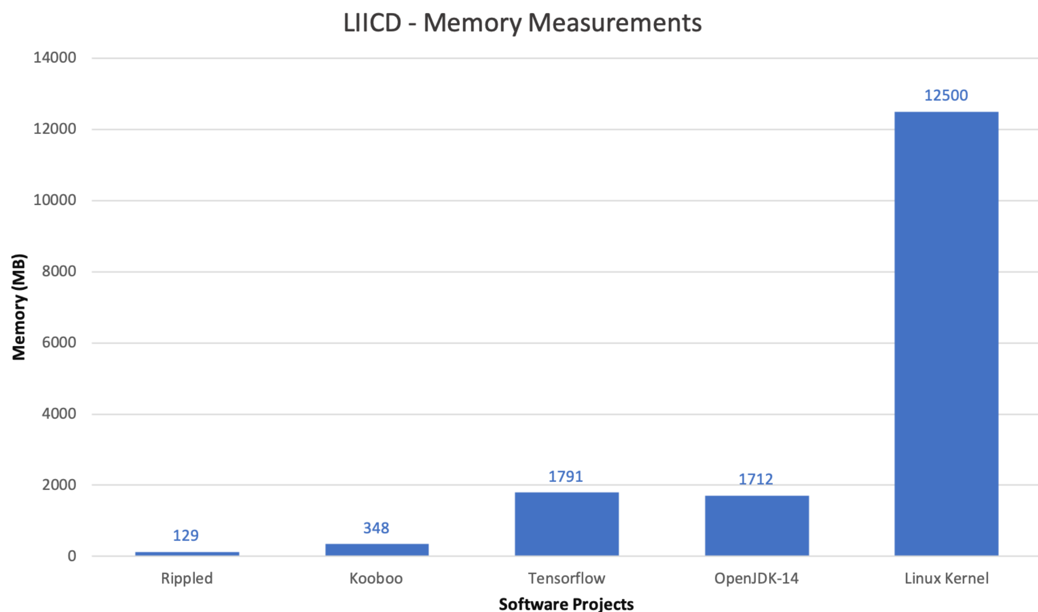

Figure 6.2: LIICD - Cumulative Memory Requirements

6.1.2 Incremental Step

The measurements for the elapsed time of incremental step flow of the LIICD implementation can be seen in Figure 6.3. As a reminder, this flow is initiated every time a code change, in the form of a commit, occurs. Our findings for the systems of Rippled, Tensorflow and OpenJDK indicate that this flow does not require more that a few seconds to

complete, on average. As for the outlier of the Kooboo experiments, low measurements like that can be justified in cases the detection process does not run at all. This can happen when the files affected by a commit do not introduce, neither remove clones, leading to the most time-consuming parts of the detection process to not execute. Lastly, for the Linux Kernel system, our experimental setup was unable to handle the memory load of this process. Note that this was not due to the memory requirements of the incremental step per se, but due to the *version control checkout* subprocess running within our application, which for large codebases requires a considerable amount of memory, larger than what was at the time available in the machine of our experimental setup.

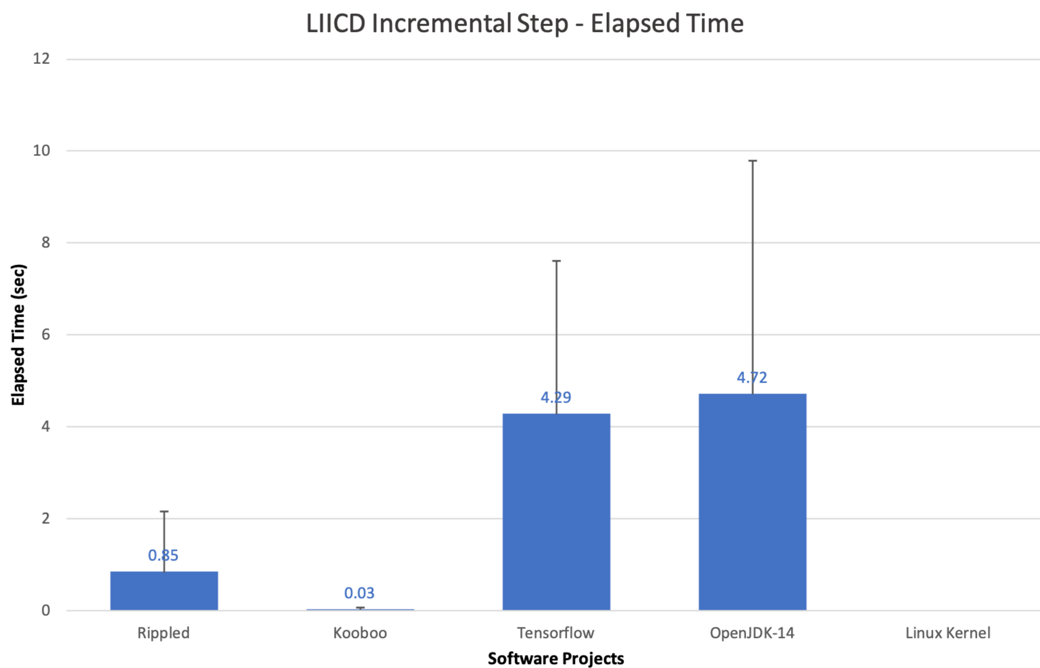

Figure 6.3: LIICD - Average Execution Time for the Incremental Step flow

On a more general note, the execution time required for the incremental step is affected by multiple different factors. The number of files included in a commit, the type of changes (creations, updates, deletions, renames), the length of the affected files, as well as the similarity threshold in the case of LSH, are all elements that could impact the results. To obtain a thorough understanding of the interplay between all these factors, additional experiments would be needed. Nevertheless, it is interesting to observe that the overall performance of this step is more efficient when compared to the index creation flow, as can be distinguished by looking at Figures 6.1 and 6.3 comparatively. This is important, considering that this flow is the one that gets executed every time a new code change occurs.

6.1.3 Clone Detection

In terms of clone detection, we ran LIICD for the four software systems for which our machine could handle the load of the incremental step. More specifically, for the sake of simplicity, we chose to analyze the 10 most recent revisions for each system and count the clones added and removed, as detected by our tool. Table 6.3 shows a general overview of the number of files affected in each commit. Although we do not specify the types of changes these file introduce (additions, updates, renames, deletions), the majority of these correspond to modifications of existing files. Table entries reported as ”0 (skipped)” correspond to commits that were not processed because they only included files that are ignored by LIICD by default (e.g. test files).

Commit No.	**# Updated files in commit**			
	Rippled	**Kooboo**	**Tensorflow**	**OpenJDK-14**
1	1	6	1	3
2	2	3	1	1
3	0 (skipped)	1	2	4
4	3	1	0 (skipped)	1
5	1	1	0 (skipped)	0 (skipped)
6	2	1	1	1
7	1	2	1	2
8	3	3	1	5
9	2	1	2	0 (skipped)
10	1	3	1	0 (skipped)

Table 6.3: Affected files per software system for each commit analyzed

Figures 6.4-6.7 show the clones that were added and removed for each commit of every software system. As a reminder, LIICD treats modified files as deletions followed by creations. This has the effect that the clones corresponding to modified files are detected as removed during the deletion substep and as newly added, during the addition substep. The latter is also depicted in our figures below, where in many cases the number of removed clones matches the number of added clones. The difference of the two numbers is the piece of information that allows us to better understand if eventually any clones were added or removed.

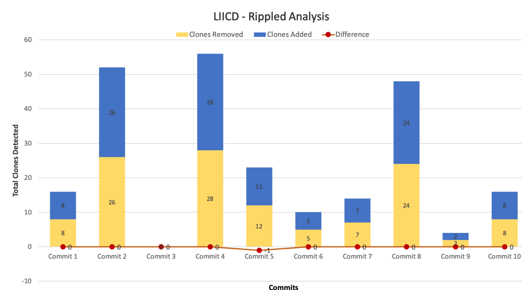

Figure 6.4: Clone Detection for Ripple

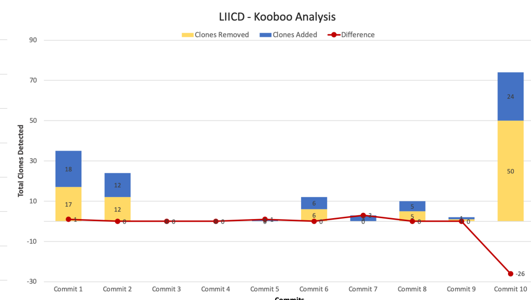

Figure 6.5: Clone Detection for Kooboo

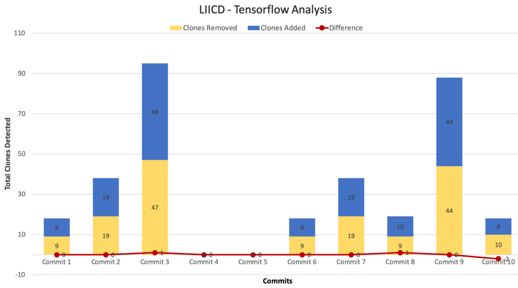

Figure 6.6: Clone Detection for Tensorflow

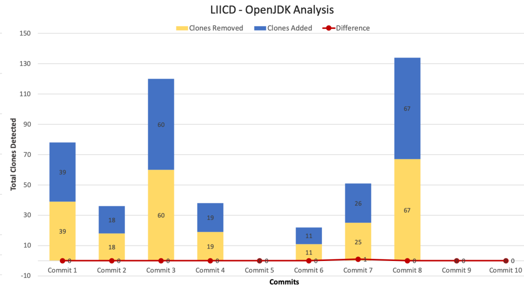

Figure 6.7: Clone Detection for OpenJDK

Another observation is that the majority of the commits do not affect the proportion of clones in the respective system's codebase. In fact, only one commit altered the total number of clones in Rippled and OpenJDK, whereas two and four commits removed/added clones in the case of Tensorflow and Kooboo, correspondingly. With respect to the number of detected clones, we further investigated the output of LIICD for commits for which the detected number of clones appears to be large (e.g. commit 8 of OpenJDK). In such cases, most of the reported clones appear to correspond to blocks of code/comments that are typically shared in many files of a system's codebase. Such an example is the fixed licence-related information, usually written at the first lines of some source files.

6.2 LIICD vs SIG Measurements

The results displayed in Table 6.4 refer to the outcome of our experiments, analyzing the performance of the overall SAT analysis along with the subprocess of clone detection embedded within the same tool. More specifically, we used SIG's SAT tool to analyze the five open-source projects constituting the dataset of this study. Since SAT is a complex tool which includes numerous underlying operations, irrelevant to code duplication detection, we isolated the relevant parts and measured the proportion of the overall elapsed time that is directly associated with clone detection.

Project	Overall SAT Analysis Time	Clone Detection Time	LIICD Index Creation & Incremental Step time
Rippled	4 min	5.63 sec	4.6 sec
Kooboo	22 min	397 sec	16.31 sec
Tensorflow	8hr 30 min	177.1 sec	91.29 sec
Openjdk-jdk14u	N/A	N/A	56.34 sec
Linux Kernel	N/A	N/A	>321.21 sec

Table 6.4: SAT measurements and overall LIICD detection time

Our experimental setup successfully managed to complete the analysis of the first three software systems, but failed for the OpenJdk and Linux systems, due to the complexity and size of these. Nevertheless, we can still observe that SAT needs a considerable amount of time to execute the analysis, even though the time needed for the detection of clones is only a small fragment of the overall analysis time. Especially for Tensorflow, the total analysis time required more than 8 hours, a number that limited the number of revisions we could analyze to 5. As for the clone detection time, the measurements range from a few seconds up to a couple of minutes for larger and more complex systems. Note though, that the detection time in the context of SAT, does not only depend on the the size of the project. The complexity of the code itself, along with the programming languages the system is written in, are some additional factors that affect this measurement, hence the reduced detection time for Tensorflow, compared to the quite smaller Kooboo system.

Although the isolated measurements might look rather small at a glance, they accumulate quite fast when the detection process runs repeatedly. Figure 6.8 allows for a better understanding of the impact of the measured times, when the clone detection process is repeated frequently. In particular, we used the Tensorflow as an example, to show how the time needed for clone detection builds up as the number of commits grows.

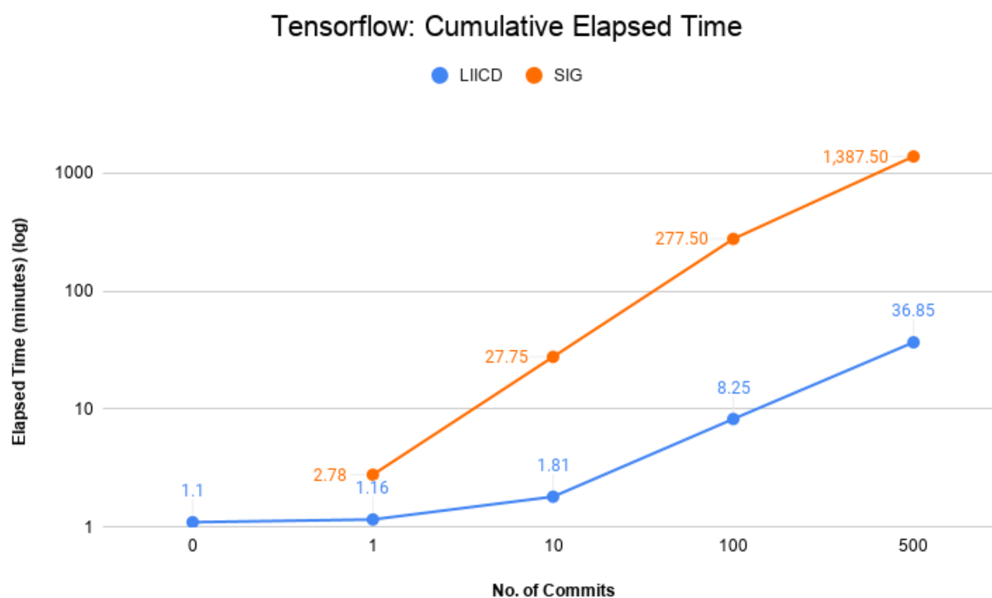

Figure 6.8: Cumulative comparison of SIG's detector versus the LIICD detector

As indicated, although the difference for a single commit does not seem large, results look different for a greater number of commits, such as 100 or 500. In the latter case for instance, using SIG's traditional detector would require approximately 1387 minutes—or about 23 hours—to run the entire analysis. On the other hand, the incremental-based approach only needs around 37 minutes for the same procedure.

6.3 LSH-based Extension Measurements

To evaluate the performance of our LSH-based implementation, we compare it with the results of the experiments of the LIICD detector. Table 6.5 shows the individual measurements for the two workflows for each of the two approaches, allowing a one-on-one comparison between them. Note that for this type of experiments we used the same experimental procedures as those described in section 6.1, meaning that the same number of LOCs for each system, as well as the same number of commits were processed.

Project	Index Creation Time (sec)		Average Incremental Step Time (sec)		Incremental Step standard deviation (sec)		Memory (MB)	
	LIICD	LSH-based	LIICD	LSH-based	LIICD	LSH-based	LIICD	LSH-based
Rippled	3.75	10.42	0.85	0.82	1.31	1.35	129	60
Kooboo	16.28	37.82	0.03	0.25	0.04	0.44	348	122
Tensorflow	87	192.8	4.29	1.57	3.32	2.13	1791	524
Openjdk-jdk14u	51.62	139.87	4.72	4.36	5.07	6.34	1712	429
Linux Kernel	321.21	954.56	N/A	4.38	N/A	10.23	12500	2600

Table 6.5: LIICD & LSH-based extension measurements

6.3.1 Index Creation

Time Measurements

Results regarding the time needed for the creation of the index, indicate a large time difference between the two implementations. This is depicted in Figure 6.9, which shows that for all the software systems in our corpus, the LSH-based extension is almost three times slower compared to the LIICD approach. As a reminder, for the LSH-based implementation we use 64 hash functions for the MinHashing process. This number, resulting in a similarity error rate of 12.5%, was purposely selected to be that low, with the goal of creating a threshold with respect to how much time this extension needs to create an index. Increasing the number of hash functions, thus decreasing the error rate, would yield additional time overhead, further growing the index creation time for the LSH-based implementation.

Figure 6.9: Execution Time for the Index Creation flow for the two implementations

Memory Measurements

Figure 6.10 illustrates the memory needs of the LSH-based extension, plotted along with our earlier measurements for the LIICD approach. Interestingly, the levels of memory usage of the LSH-based approach were found to be, in most cases, two to three times lower than those of the LIICD detector. Especially in the case of the Linux Kernel, there is a substantial difference between the two approaches, with the LIICD requiring about five times more memory to complete the detection.

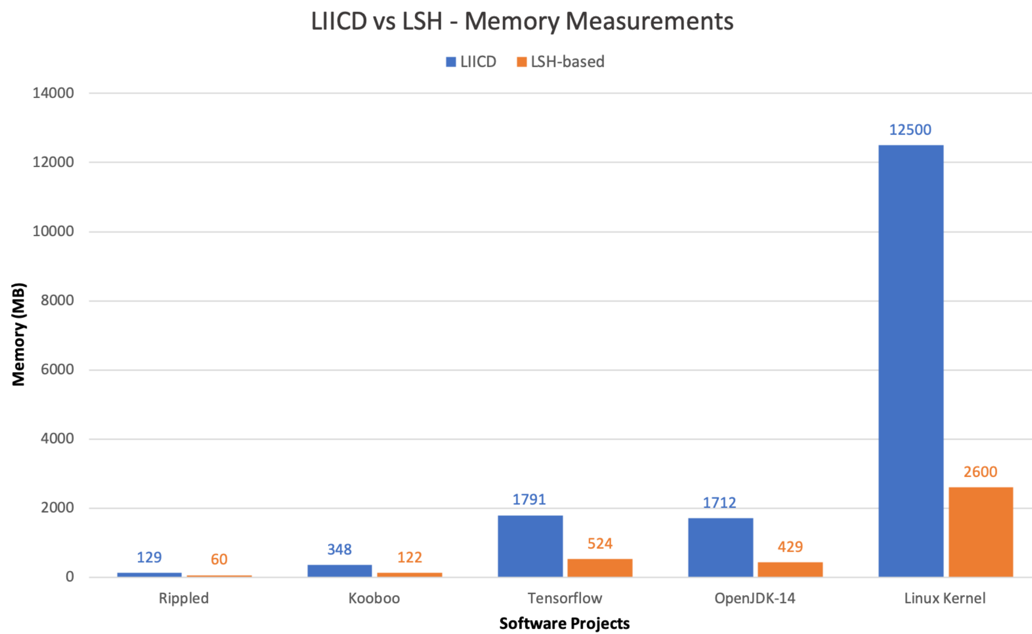

Figure 6.10: Memory requirements for the two implementations

6.3.2 Incremental Step

Moving on to the performance of the incremental step for the two implementations, Figure 6.11 illustrates the corresponding measurements. At first glance, the respective findings look confusing. That is because we would expect a worse performance for all the projects analyzed by the LSH-based implementation, considering the extra step of calculating the clone index entries on the fly, compared to the LIICD approach. However, additional investigation showed that the reason why this is not depicted in our visual is because, in many cases, the selected similarity threshold of 20% was not low enough to allow for many files to be identified as similar. As a result, many detections are skipped, resulting in the decreased time shown in our diagram.

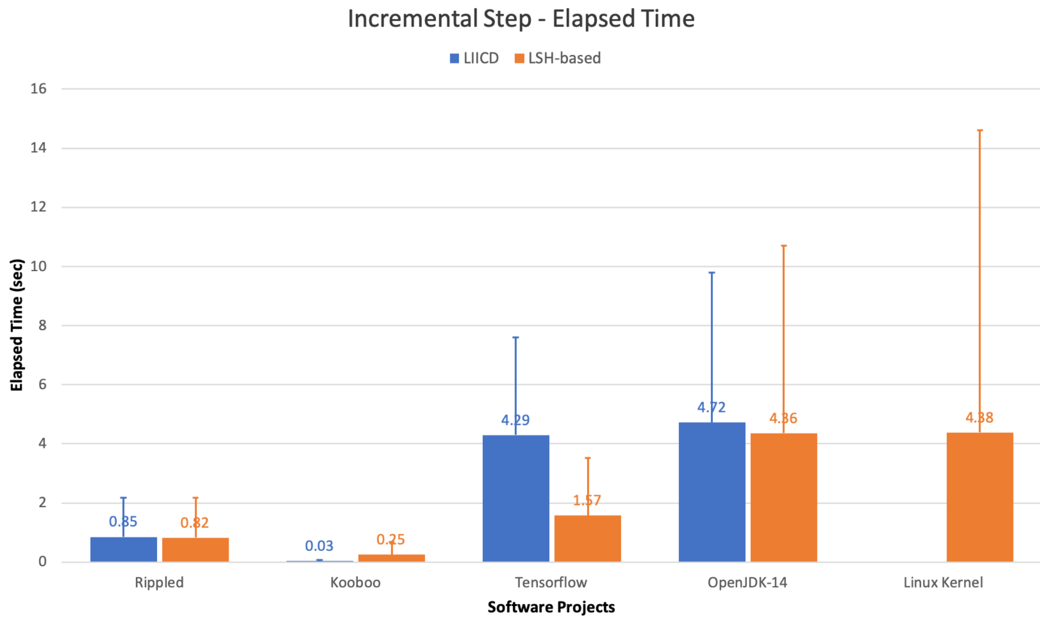

Figure 6.11: Execution Time for the Incremental Step flow for the two implementations

Further investigation at the reasons behind this large difference between the two implementations, indicates that the MinHashing part of the overall LSH scheme is quite computationally heavy. In fact, as shown in Figure 6.12, isolated time measurements of the index creation step for the LSH-based implementation using the Tensorflow system as input, show that approximately 37% of the index creation time was spent on the substep of MinHashing. The remaining 63% was split among the Shingling and the process of banding (shown as LSH), with the former requiring the most time.

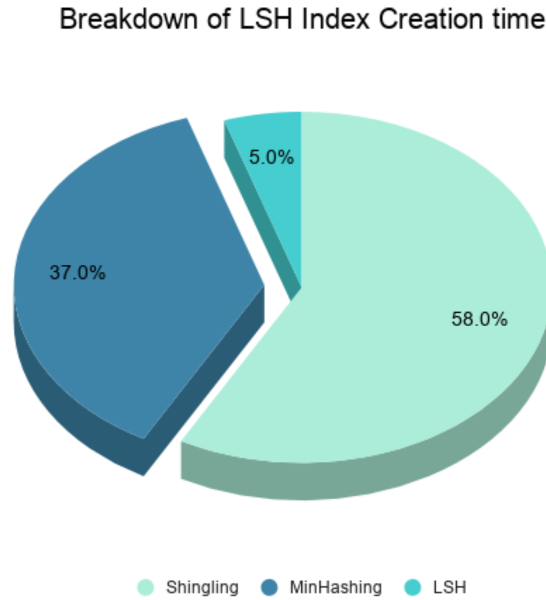

Figure 6.12: Index Creation sub-measurements for the LSH-based implementation

6.3.3 Varying Hash Functions

In section 6.3.1, we mentioned that for the purposes of our experiments with the LSH-based implementation, we used 64 hash functions for the process of MinHashing. Additionally, it is interesting to explore how alterations in the number of hash functions, affect the incremental step and overall index creation time, considering that MinHashing is one of the most time-consuming parts of the LSH-based implementation, Figure 6.13 shows the implications of increasing the number of hash functions—hence decreasing the error rate—on the time needed to create the corresponding index and the incremental step processing time. More specifically, we used Kooboo and measured the index creation time for 64, 128, 256 and 512 hash functions, which in turn correspond to an error rate of 12.5%, 8.8%, 6.25% and 4.4%, respectively.

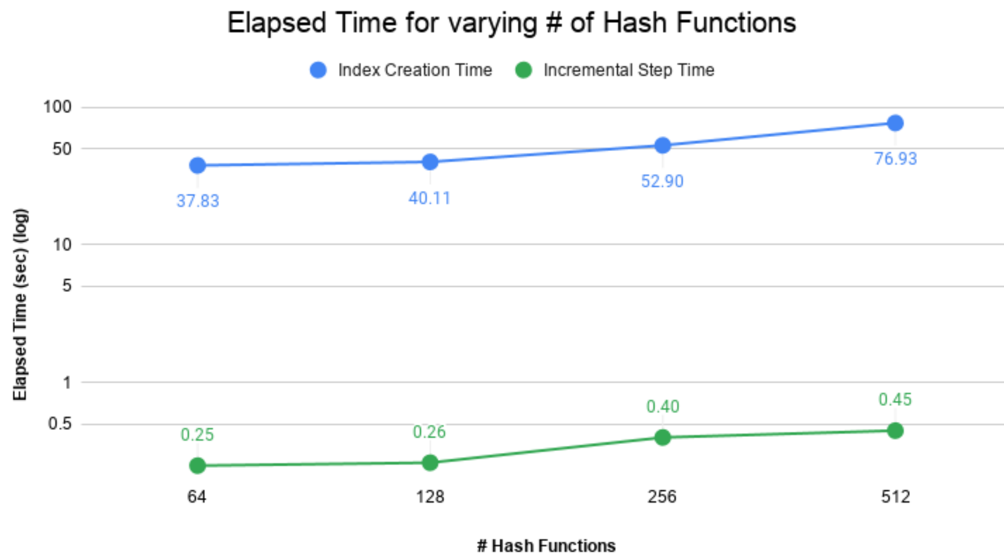

Figure 6.13: Execution Time implications for varying number of hash functions during MinHashing

As shown, the number of hash functions plays an important role to the time needed to process the two workflows. In particular, looking at the measurements of the lowest threshold of 64, and the highest one of 512 hash functions, we can see that the index creation time almost doubled. The behavior is consistent with the incremental step flow measurements, where an increase of similar magnitude was observed.

Chapter 7

Discussion

In this chapter we interpret the results of our experiments and discuss their implications. Furthermore, we discuss the threats to validity for this study and provide suggestions for practitioners, with a closer eye to a potential usage of the proposed detectors within SIG.

7.1 Main Findings

The section reports the conclusions that can be inferred by examining the results of our experiments. More specifically, we analyze the outcome of our experiments and discuss how these correspond to our initial intuition.

7.1.1 LIICD Approach

The conclusions that can be drawn from our experiments vary. Firstly, the findings indicate that the LIICD detector is faster than expected with respect to measurements reported by *Hummel et al.* [24]. In particular, LIICD's index creation time measurements show that the approach scales pretty well considering that the index for a system of 20 million LOCs took a bit over five minutes to create. Additionally, our measurements for the incremental step flow indicate that the particular operation runs in a matter of few seconds. The latter is crucial taking into account that this flow is initiated frequently, every time a new code change in the form of a commit occurs. Although additional experiments are needed to identify how exactly the factors affecting this flow's performance actually influence it, our experiments already allow us to conclude that this step runs efficiently.

In terms of memory, our experiments showed that the memory usage of our LIICD approach is generally low for today's standards, but can quickly escalate when larger systems are analyzed. In the case of the Linux Kernel, the memory usage peaked at 12.5GB, a considerable number, despite the fact that such systems are the exception rather than the rule, in terms of codebase size. As for the failure of LIICD to run the incremental step for the same project due to insufficient memory, this is not a matter of concern, since that was caused by the version control revision checkout operation that runs within our application. In a real world scenario, where commits are not simulated, this operation would not run, eliminating thus any memory usage spikes caused by this.

7.1.2 LIICD vs SIG

The results of our analysis using SIG's internal SAT tool provide some valuable insights. First of all, we observed that SAT shows shortcomings when larger systems are given as input. The analysis of Tensorflow required more than 8 hours to complete, whereas SAT was unable to analyze larger and more complex codebases, such as that of OpenJdk and Linux, despite the experiments being run on a high-spec machine. Even though such large systems are not that many, this is still a limitation of SIG's tool.

Additional sub-measurements of the clone detection time within SAT showed that this process takes a very small proportion of the overall analysis time. However, comparing these measurements with those of LIICD's experiments through a series of commit analyses, results in some interesting findings. In particular, although the measurements themselves look small when it comes to the analysis of a single system revision, the accumulated elapsed time grows rapidly as more and more revisions need to be analyzed. This would not be a problem if the frequency of incoming commits was low, but in today's era where tens or even hundreds of commits are pushed daily, it becomes apparent how a traditional detector, such as the one SIG is using, is impractical in an incremental setting.

7.1.3 LIICD vs LSH-based Extension

Lastly, regarding our experiments evaluating the efficiency of our LSH-based extension, we can again extract some useful information. More specifically, the index creation step of our LSH-based approach was found to be two, and in some cases three times slower compared to the LIICD implementation. Potential reasons causing this, can be found in the complexity of the MinHashing operation which constitutes a significant proportion of the overall LSH scheme. This becomes obvious considering that during MinHashing, every shingle for every set of shingles has to be hashed by the predefined k hash functions. On the contrary, from the perspective of memory usage, LSH was found to be much more efficient, resulting in a decrease of up to five times in the case of Linux. However, this alone does not justify the use of this approach instead of LIICD.

Additionally, the measurements for the incremental step did not match our initial intuition. In particular, we expected the incremental step flow of the LSH-based implementation to be slower compared to that of LIICD, due to the calculation of the index entries on-the-fly. However, our findings show contrasting results. Practically, this can be justified by the similarity threshold that we used, which, as it turns out, did not result in many matches between the source files. To better understand its behavior, further investigation between the relation of the similarity threshold and the execution time needed for the incremental step flow of the LSH-based implementation, is needed.

Another interesting finding is that the size of a software system is not a good indicator to forecast the time required for the incremental step workflow. Apparently, this time is affected by factors such as (1) the number of files in a commit, (2) the type of changes and (3) the length (in LOCs) of these files. Future work should further investigate the behavior of this workflow against these variables.

Finally, the measurements for the incremental step of the two implementations show

that the LIICD approach performs better. As for the incremental step, additional experiments for the LSH-based implementation are required to be able to deduct more insightful conclusions. However, any such attempt would be futile without first investigating whether the execution time of the index creation step of the LSH-based implementation, can be reduced to better or similar levels as those of the LIICD approach.

7.2 Implications

The results of our experiments for the LIICD detector are not completely aligned with those mentioned in the original paper by *Hummel et al.* [24]. More specifically, we observed a considerable difference to the time measurements for the index creation and incremental step workflows. Naturally, minor deviations were expected due to three major factors affecting our measurements. These are:

1. **Experimental Setup:** The hardware used for our experiments is faster compared to the one used in the original study by *Hummel et al.* [24].

2. **Normalization:** To account for language independence, our implementation eliminates the tokenization step of the original detector, removing the additional overhead that this process requires.

3. **In-memory persistence:** The measurements mentioned in the original study refer to an implementation that persists the intermediate information in a database. In our work, we use in-memory persistence.

Considering these factors, we anticipated a decrease in the time needed for the index creation. Although this expectation is supported by our findings, the difference is much larger than these factors could justify. In particular, the original study mentions 7 hours and 4 minutes as the time needed to create the index for a project of approximately 40 million LOCs. However, in our experiment with the Linux Kernel, a project half that size, the index creation took a little over 5 minutes for LIICD and about 15 for the LSH-based approach, indicating approximately a 42 and 14 factor difference in the processing time respectively, assuming that Hummel's approach scales linearly. This observation alone suggests that the index creation is much faster than expected, making thus any attempt to further improve it rather challenging.

As for the incremental step, the findings provide new insights when compared with the original study. More specifically, our reported measurements allow us to conclude that the overall incremental step time is affected by the system's size. This is not logical at first, considering that querying and updating the hash-based index should result in similar constant-time measurements. However, for large systems where the index is populated with a very extensive number of entries, collisions might occur, adding extra overhead to the overall calculation. Furthermore, multiple factors such as the number of files in a commit, the length of these files, as well as the type of the changes these introduce, also affect the individual measurements. Lastly, the original study simulates commits by randomly

removing and re-adding source files. Our findings allow for a better understanding of the behaviour of the incremental step workflow, since in our case, we used actual commits.

7.3 Threats to Validity

7.3.1 Internal Validity

SAT Measurements

In this study, we used SIG's SAT tool to compare the performance of the proposed incremental clone detector, with that of the traditional detector embedded within SAT. However, SAT is a complex tool consisting of multiple different components. Some of these are related to clone detection, but others are only used for the calculation of other maintainability metrics. Furthermore, isolating the operations associated with clone detection is not a simple task due to the complexity and large number of substeps within SAT. Nevertheless, although we carefully selected the subset of operations for which we measured the elapsed time, there might be additional sub-steps that have not been considered. However, we do not expect these steps, if any, to contribute much in the measured elapsed time, thus invalidate our findings.

Clone Coverage

In terms of clone coverage for the LIICD detector (i.e. whether our tool correctly detects all scenarios under which a clone might occur), additional experiments, utilizing benchmarks, would be ideal to verify this extensively. In our work, we manually tested this by crafting commits that simulate multiple use cases of adding, modifying, renaming or deleting a file in the context of a commit. Nevertheless, there might be corner cases that we did not consider and therefore did not test our detector for.

7.3.2 External Validity

The dataset of software systems used in our experimental design, only consists of five software systems. Although we selected these to deliberately differ in codebaze size and the programming languages they use, new findings might occur by experimenting with a larger corpus of systems.

7.4 Applicability within SIG

In this section, we examine the applicability of the proposed detectors in the context of SIG. More specifically, we first dive into the potential usage of these detectors and then discuss some important points that would need to be considered, if either of these detectors was to be part of SIG's collection of tools.

7.4.1 Potential Usage

SIG's analyses models include the ability to measure maintainability for code changes rather than for entire codebases utilizing the Delta Maintainability Model (DMM) [14]. This model allows to incrementally measure code changes and rank their code quality, instead of measuring the impact of code changes towards the entire system's maintainability. However, with respect to clone detection, DMM can only detect clones found within the affected files of a particular commit. This is not optimal, since an addition in File A might introduce a clone with File B and in case the latter is not part of the files affected by the specific commit, the clone will be missed.

Both implementations introduced in this study could potentially be used to overcome this issue, either as an external tool that the DMM uses or even embedded within DMM's implementation. In fact, both support the detection of clones outside a commit's context, resolving DMM's related limitation. Additionally, the introduced detectors are also language-independent and are capable of detecting Type-1 clones, two requirements essential for SIG's needs. Lastly, there is also the potential to detect Type-2 clones with the addition of a normalization step. In this case however, a language-specific parser would be necessary, removing the language-agnostic characteristic of these detectors, scope of this thesis work.

7.4.2 Considerations

Clones Snapshot

One of the characteristics of the proposed incremental detectors is that they only output the clones that were added or removed at each code revision and not an entire snapshot of all the clones in a codebase. For the latter to be possible, there are two potential options. First, the introduction of a clone management logic. This, would allow the aggregation of clones by looking at those that were added and removed, from the beginning of the codebase (or a revision where all the clones were known), up to the revision of interest. Secondly, another way to achieve the same result is through the introduction of an option that allows the querying of the index using every single file in the codebase. However, such a solution would be extremely time-demanding, considering that all the files of a system would have to go through the detection pipeline mentioned earlier in this study.

Data Persistence

Each of the proposed detectors utilizes intermediate information, which is used across the revisions to detect clones. The existing implementations store this information in memory. However, in a real-world setup, a database would be more appropriate for this purpose. With the current setup, the applications need to always be on hold, otherwise the calculated index will be lost. On the contrary, this would not be a requirement with hard-disk persistence, with which the index entries could be fetched at any point in time.

Output Format

During the development of the discussed detectors, we did not consider any potential integration with SIG's existing or future tools, in which case we would have to investigate how such tools could be interfaced with our implementations. Consequently, the format of the output has not been designed in a way so that it can be seamlessly consumed by these, thus further changes would be necessary to account for that.

Comments Removal

Lastly, the removal of comments could also be a potential desirable feature. This, would require additional preprocessing of the codebase of a system and possibly its segmentation into technologies (programming languages) so that different commenting styles can be handled appropriately. In the context of SIG, processes that remove code comments are already available and could be utilized to achieve this. Therefore, in this context, comments removal is only a matter of integrating these processes with the discussed detectors.

Chapter 8

Conclusion & Future Work

In this final chapter we discuss the conclusions that can be drawn after the completion of this study and retrospectively relate them to our initial research questions. Furthermore, we provide suggestions for future work, allowing for further investigation of the potential of the LSH-based approach.

8.1 Conclusion

The need to tackle the maintainability issues that come with code duplication, combined with the desire to run the clone detection process repeatedly, brought the appearance of incremental clone detection techniques. Many such techniques exist, however most of them require language parsers, a component that automatically eliminates the feature of language independence for the corresponding proposed clone detectors. Consequently, programming languages for which finding or creating a parser is challenging, cannot be analyzed in the context of clone detection.

In this study, our aim was to identify a way to build a language-agnostic clone detector and investigate what kind of information can be stored, so that this detector works incrementally. In that aspect, we modified *Hummel's* original algorithm and examined the use of LSH in an attempt to extend and improve the original approach. Our findings showed that the intermediate representation of a modified Clone Index can indeed be used to achieve language independence, while also satisfying the requirements set at the beginning of this study. Furthermore, our experiments revealed that the LIICD approach was found to be much faster that expected, leaving limited room for further improvement. The latter is verified by observing the performance of the LSH-based implementation, which at its discussed form, performs worse. All things considered, in the context of this thesis, we successfully built a language-agnostic incremental clone detector based on the approach by *Hummel et all.* [24] and extended this utilizing LSH. However, further research is needed to investigate the potential of the LSH-based approach, in the context of the original algorithm.

Additionally, another goal of this study was to discover whether such an incremental approach performs adequately compared to a commercial-grade traditional detector, such as the one embedded within SIG's SAT tool. For that purpose, we ran SAT for every system

in our dataset and measured the time needed for clone detection. Although SAT was unable to process the largest and most complex systems of our corpus, we were still able to extract results from the analysis of the smaller ones. In comparison with the measurements of the incremental detector, these results indicate a considerable improvement in the accrued clone detection time when the detection process is repeated for a series of commits. Consequently, we conclude that the adoption of such an approach by SIG would be beneficial in terms of time and resource efficiency. Furthermore, the capability of our detectors to discover clones outside of a commit's context sets the ground for a potential future integration of these, in the scope of the Delta Maintainability Model [14].

8.2 Future Work

One line of future research is to further investigate ways to improve the suitability of Locality Sensitive Hashing as an extension of the incremental detector proposed at the beginning of this study.

According to the discussed measurements, MinHashing is a rather time-expensive operation of the specific LSH scheme used in this study. During this step, every shingle of each shingle-set has to be hashed by k hash functions, a process rather demanding. However, further research in the area of LSH has yielded approaches that have attempted to eliminate this bottleneck. Related studies such that of *SuperMinHash* [17] and *Li et al.* [35] claim to be capable of achieving identical behaviour with a single hash function. Such an optimization could prove to be a significant improvement to the time needed for the creation of the corresponding index of the LSH-based approach and would be certainly worth trying.

The LSH-based implementation, by nature, can lead to decreased recall. In this implementation, the clone detection process runs only for files that were found to be similar based on the defined similarity threshold. Therefore, it would be interesting to investigate the proportion of the missed clones, while experimenting with varying similarity thresholds. However, such additional measurements, concerning the incremental step flow of the LSH-based approach, would only be useful if the time needed for the index creation step could be brought down to levels comparable with those of the LIICD approach.

One of the factors affecting the performance of any LSH scheme is the number of input elements the method is applied on. In this work, this corresponds to the number of source files in a software system's codebase. That considered, measuring and comparing the performance between projects with similar size—in terms of LOCs—but with a different number of source files, could help better understand the impact of this factor on the method's performance.

Bibliography

[1] Akshat Agrawal and Sumit Kumar Yadav. A hybrid-token and textual based approach to find similar code segments. In *2013 fourth international conference on computing, communications and networking technologies (ICCCNT)*, pages 1–4. IEEE, 2013.

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

[3] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[4] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

[5] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.

[6] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[7] Frank Buschmann. To pay or not to pay technical debt. *IEEE software*, 28(6):29–31, 2011.

[8] Raffaele Cappelli, Matteo Ferrara, and Davide Maltoni. Fingerprint indexing based on minutia cylinder-code. *IEEE transactions on pattern analysis and machine intelligence*, 33(5):1051–1057, 2010.

[9] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[10] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[11] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[12] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280, 2007.

[13] Michiel De Wit, Andy Zaidman, and Arie Van Deursen. Managing code clones using dynamic change tracking and resolution. In *2009 IEEE International Conference on Software Maintenance*, pages 169–178. IEEE, 2009.

[14] Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. The delta maintainability model: measuring maintainability of fine-grained code changes. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 113–122. IEEE, 2019.

[15] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 109–118. IEEE, 1999.

[16] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. Distributed representations of tuples for entity resolution. *Proceedings of the VLDB Endowment*, 11(11):1454–1467, 2018.

[17] Otmar Ertl. Superminhash-a new minwise hashing algorithm for jaccard similarity estimation. *arXiv preprint arXiv:1706.05698*, 2017.

[18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[19] Marco Funaro, Daniele Braga, Alessandro Campi, and Carlo Ghezzi. A hybrid approach (syntactic and textual) to clone detection. In *Proceedings of the 4th International Workshop on Software Clones*, pages 79–80. ACM, 2010.

[20] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[21] Nils Göde and Rainer Koschke. Incremental clone detection. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 219–228. IEEE, 2009.

[22] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007.

[23] Yoshiki Higo, Ueda Yasushi, Minoru Nishino, and Shinji Kusumoto. Incremental code clone detection: A pdg-based approach. In *2011 18th Working Conference on Reverse Engineering*, pages 3–12. IEEE, 2011.

[24] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: incremental, distributed, scalable. In *2010 IEEE International Conference on Software Maintenance*, pages 1–9. IEEE, 2010.

[25] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

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

[27] J Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM*, volume 94, pages 120–126, 1994.

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

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

[30] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pages 40–56. Springer, 2001.

[31] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.

[32] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[33] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2nd edition, 2014. ISBN 1107077230.

[34] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[35] Ping Li, Art Owen, and Cun-Hui Zhang. One permutation hashing for efficient search and learning. *arXiv preprint arXiv:1208.1259*, 2012.

[36] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[37] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[38] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM, 2006.

[39] Andrian Marcus and Jonathan I Maletic. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114. IEEE, 2001.

[40] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.

[41] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18, 2013.

[42] Tung Thanh Nguyen, Hoan Anh Nguyen, Jafar M Al-Kofahi, Nam H Pham, and Tien N Nguyen. Scalable and incremental clone detection for evolving software. In *2009 IEEE International Conference on Software Maintenance*, pages 491–494. IEEE, 2009.

[43] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in java software systems. *Empirical Software Engineering*, 24(3):1499–1537, 2019.

[44] Chaiyong Ragkhitwetsagul and Jens Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, pages 1–49, 2019.

[45] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[46] Chanchal K Roy and James R Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166. IEEE, 2009.

[47] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[48] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[49] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[50] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[51] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[52] Anshumali Shrivastava and Ping Li. In defense of minhash over simhash. In *Artificial Intelligence and Statistics*, pages 886–894, 2014.

[53] Sadhan Sood and Dmitri Loguinov. Probabilistic near-duplicate detection using simhash. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1117–1126, 2011.

[54] Jeffrey Svajlenko and Chanchal K Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140. IEEE, 2015.

[55] Vasilis Verroios and Hector Garcia-Molina. Top-k entity resolution with adaptive locality-sensitive hashing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1718–1721. IEEE, 2019.

[56] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[57] Kunpeng Zhang, Shaokun Fan, and Harry Jiannan Wang. An efficient recommender system using locality sensitive hashing. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[58] Wei Zhou, Jiankun Hu, and Song Wang. Enhanced locality-sensitive hashing for fingerprint forensics over large multi-sensor databases. *IEEE Transactions on Big Data*, 2017.

Appendix A

Corpus Collection

This appendix provides information regarding the status of each of the software systems in our dataset during the time of our analysis.

A.1 Project Snapshots

Table A.1 shows the status of the git-based repository of every open source project we used, during the time of our analysis. More specifically, it points out the main branch that was used along with information about the most recent commit for the particular branch. Starting from the reported commits for each system and going back 50 commits in each case—excluding merges—, one can retrieve the software system revisions analyzed in our study.

Project	Branch	HEAD commit id	HEAD commit meta-info
Rippled	develop	cd78ce3	Author: Carl Hua Date: Tue Apr 14 09:50:15 2020 -0400 Add PR automation for project boards
Kooboo	master	39406b7	Author: CN name Date: Mon Jun 15 09:58:57 2020 +0800 fixed monaco bug
Tensorflow	master	f926d8c	Author: A. Unique TensorFlower Date: Mon Jun 15 03:38:44 2020 -0700 Introduces a new experimental package that: - Defines a schema for configuring delegates - Defines a C++ plugin mechanism using the schema...
Openjdk-jdk14u	master	e23aaed	Author: Prasadrao Koppula Date: Thu Jun 11 21:54:51 2020 +0530 8246031: SSLSocket.getSession() doesn't close...
Linux	master	9cb1fd0	Author: Linus Torvalds Date: Sun May 24 15:32:54 2020 -0700 Linux 5.7-rc7

Table A.1: Project snapshots at the time of analysis

Appendix B

Excluded Directories & Files

Table B.1 shows the directories and file extensions ignored by both of our implementations. This happens at two different phases, during the initial parsing of the codebase and during the parsing of the changes included in a commit.

EXLUDED DIRS	node_modules, assets, build, classes, gradle, licenses, icu, dcn21, fixtures, docs, test, tests, examples, changelogs
EXLUDED FILES	.3dm, .3ds, .3g2, .3gp, .7z, .a, .aac, .adp, .ai, .aif, .aiff, .alz, .ape, .apk, .ar, .arj, .asf, .au, .avi, .bak, .baml, .bh, .bin, .bk, .bmp, .btif, .bz2, .bzip2, .cab, .caf, .cgm, .class, .cmx, .cpio, .cr2, .cur, .dat, .dcm, .deb, .dex, .djvu, .dll, .dmg, .dng, .doc, .docm, .docx, .dot, .dotm, .dra, .DS_Store, .dsk, .dts, .dtshd, .dvb, .dwg, .dxf, .ecelp4800, .ecelp7470, .ecelp9600, .egg, .eol, .eot, .exe, .f4v, .fbs, .fh, .fla, .flac, .fli, .flv, .fpx, .fst, .fvt, .g3, .gh, .gif, .epub, .graffle, .gz, .gzip, .h261, .h263, .h264, .icns, .ico, .ief, .img, .ipa, .iso, .jar, .jpeg, .jpg, .jpgv, .jpm, .jxr, .key, .ktx, .lha, .lib, .lvp, .lz, .lzh, .lzma, .mng, .lzo, .m3u, .m4a, .m4v, .mar, .mdi, .mht, .mid, .midi, .mj2, .mka, .mkv, .mmr, .mobi, .mov, .movie, .mp3, .mp4, .mp4a, .mpeg, .mpg, .mpga, .mxu, .nef, .npx, .numbers, .nupkg, .o, .oga, .ogg, .ogv, .otf, .pages, .pbm, .pcx, .pdb, .pdf, .pea, .pgm, .pic, .png, .pnm, .pot, .potm, .potx, .ppa, .ppam, .ppm, .pps, .ppsm, .ppsx, .ppt, .pptm, .pptx, .psd, .pya, .pyc, .pyo, .pyv, .qt, .rar, .ras, .raw, .resources, .rgb, .rip, .rlc, .rmf, .rmvb, .rtf, .rz, .s3m, .s7z, .scpt, .sgi, .shar, .sil, .whl, .sketch, .slk, .smv, .snk, .so, .stl, .suo, .sub, .swf, .tar, .tbz, .tbz2, .tga, .xlam, .tgz, .thmx, .tif, .tiff, .tlz, .ttc, .ttf, .txz, .udf, .uvh, .uvi, .uvm, .uvp, .uvs, .uvu, .viv, .vob, .war, .wav, .wax, .wbmp, .wdp, .weba, .webm, .webp, .wim, .wm, .wma, .wmv, .wmx, .woff, .woff2, .wrm, .wvx, .xbm, .xif, .xla, .xls, .xlsb, .xlsm, .xlsx, .xlt, .xltm, .xltx, .xm, .xmind, .xpi, .xpm, .xwd, .xz, .z, .zip, zipx, .txt, .md, .bat, .jks, .sh, .prpt, .ini, .db, .plist, .ver, .pb, .data-00000-of-00001, .index, .golden, .pbtxt.gz, .mdb, .meta, .bytes, .lite, .h5, .data-00000-of-00002, .data-00001-of-00002, .map, .elf, .skb, .skp, .dtbo, .mat, .dll, .Rascal, .exr, .blend, .pfb, .xcf, .odg, .out, .sgml, .pfx, .fig, .mo, .install

Table B.1: Excluded Directories & File Extensions