Towards Semantic Clone Detection, Benchmarking, and Evaluation

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Farouq Al-omari

Permission to Use

In presenting this in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my work or, in their absence, by the Head of the Department or the Dean of the College in which my work was done. It is understood that any copying or publication or use of this or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my .

Disclaimer

Reference in this to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building, 110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9 Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

Developers copy and paste their code to speed up the development process. Sometimes, they copy code from other systems or look up code online to solve a complex problem. Developers reuse copied code with or without modifications. The resulting similar or identical code fragments are called code clones. Sometimes clones are unintentionally written when a developer implements the same or similar functionality. Even when the resulting code fragments are not textually similar but implement the same functionality they are still considered to be clones and are classified as semantic clones. Semantic clones are defined as code fragments that perform the exact same computation and are implemented using different syntax.

Software cloning research indicates that code clones exist in all software systems; on average, 5% to 20% of software code is cloned. Due to the potential impact of clones, whether positive or negative, it is essential to locate, track, and manage clones in the source code. Considerable research has been conducted on all types of code clones, including clone detection, analysis, management, and evaluation. Despite the great interest in code clones, there has been considerably less work conducted on semantic clones.

As described in this thesis, I advance the state-of-the-art in semantic clone research in several ways. First, I conducted an empirical study to investigate the status of code cloning in and across open-source game systems and the effectiveness of different normalization, filtering, and transformation techniques for detecting semantic clones. Second, I developed an approach to detect clones across .NET programming languages using an intermediate language. Third, I developed a technique using an intermediate language and an ontology to detect semantic clones. Fourth, I mined Stack Overflow answers to build a semantic code clone benchmark that represents real semantic code clones in four programming languages, C, C#, Java, and Python. Fifth, I defined a comprehensive taxonomy that identifies semantic clone types. Finally, I implemented an injection framework that uses the benchmark to compare and evaluate semantic code clone detectors by automatically measuring recall.

Acknowledgements

First, I would thank Almighty Allah, who gave me the courage, power and patience to start and complete the journey of Ph.D..

I would like to express my deepest heartfelt thanks to my supervisor Dr. Chanchal K. Roy. I got endless support, guidance, patience and care from Dr. Roy in the past few years. It was impossible to complete this work without him.

A great thanks to Dr. Kevin A. Schneider, who supervised me during my last semester in the Ph.D. Then, my sincere thanks to my Ph.D. committee, Dr. Gord McCalla, Dr. Shahedul Khan and Dr. Mark Keil for their support with guidance, comments and suggestions. Many thanks to my external examiner Dr. Emad Shihab for his comments and suggestions.

I would like to thank the co-authors of my publications, Dr. Chanchal Roy, Dr. Juergen Rilling, Iman Keivanloo and Tonghao Chen.

I am grateful to all the Department of Computer Science staff for their advice, help and support. I especially thank Gwen Lancaster, Heather Webb and Sophie Findlay.

I would like to thank all my lab members for their support and valuable discussions. Especially, Md. Saidur Rahman for his support and advises.

I am grateful to my father Ahmad Al-omari and my mother Amneh Alomari for their endless love and prayers. My thanks to my lovely wife, who looked after our kids and gave me time, support, love and advice. My heart pieces, my beautiful kids, Ahmad, Omar, Raghad, Batool and Ali, Thank you all. Also, I am thankful to my brothers Jafaar, Noumaan, Mohammad and Hisham and my sisters Buthina, Rahma and Rodina for their love and inspiration.

Many thanks to all my friends, to everyone's support and help. Particularly, my second families Mary Rowan and Mark De Jong, Christina and Ilias, and Bart and Heather.

I dedicate this thesis to my mother Amneh Alomari, my father Ahmad Al-omari, and my beloved wife Najah Alomari, for their endless love, care and support.

Contents

List of Tables

List of Figures

List of Abbreviations

CIL	Common Intermediate Language
IL	Intermediate Language
LOC	Lines of Code
TCM	Total Cloned Methods
TCLOC	Total Cloned Line of Code
FAWC	File Associated with Clones
CCR	Clone Class Radius
NOF	Number of Files
NOM	Number of Methods
NOCC	Number of Clone Classes
AST	Abstract Syntax Tree
PDG	Program Dependence Graph

Chapter 1

Introduction

Software clones are defined as similar (near-miss) or identical (exact) code fragments in terms of syntax or semantics. Usually, these code fragments result from the practice of programmers copying and pasting code, which produces identical clones. However, if the copied code fragments have minor modifications, they result in near-miss clones. Significant modifications to cloned code may result in the code no longer being considered a clone. Conversely, some clones are unintentionally introduced into software systems when programmers implement a common task or when they use a library or API to implement the same or similar functionality [10, 141]. When two code fragments have the same functionality and are implemented using different syntax, they are called semantic clones.

Software code cloning offers benefits during the development process.Usually, developers reuse their own code to save the time of rewriting it, or they reuse others' code to overcome some programming and design limitations [141]. Skilled developers pay more attention in order to choose higher quality, well tested, and bug-free code to clone [33, 91]. On the other hand, the cloned code might have a serious problem, i.e. bugs that need more testing or updates in the maintenance phase [57, 134, 135, 79]. Practitioners have two different opinions about whether clones are harmful [23, 44, 83, 82, 110, 122] or not [86, 33, 83, 85, 59, 33]. As a result, some studies target software clone harmfulness/usefulness [67, 86]. For example, [91, 58, 113] compared the co-changes of cloned to non-cloned code. Other studies compared the stability of cloned and non-cloned code [68, 62, 123, 99].

Over the decades, practitioners have proposed different techniques to detect both syntactic and semantic clones. Detecting syntactic clones is easier than semantic clones. In syntactic clone detection, the source code is normalized then transformed into other representations (token, tree, or vectors) before it is used for comparison [89, 118, 27, 104, 170, 45]. However, in semantic clone detection, more normalization needs to be done. For instance, dependencies and relationships, perhaps using a Program Dependence Graph (PDG), have to be identified and represented, and the functionality should be captured and used in comparing code units [178, 151]. Tree and graph comparison algorithms are well known as compute-extensive [131, 53, 38, 66, 65, 148, 35]. A recent study shows that existing techniques and tools have some limitations in the detection of semantic clones (functional clones) [167]. Compute efficient and accurate semantic clone detection has been a big challenge in the area of code clones.

More recently, there has been an ongoing trend towards multi-language software development to take

advantage of different programming languages [95, 29, 16]; specifically in the .NET context. For multi-language development, two key usage scenarios can be distinguished: (1) combining different programming languages within a single, often large and complex system, and (2) the use of several languages for re-implementation of a current system to support a new client, application, or due to non-technical reasons. As a result, the ability to detect and manage similar code reuse patterns that might exist in these multi-language systems becomes essential. While many clone detection tools are capable of supporting different programming languages, they lack actual cross-language support during detection time. Consequently, these tools only detect clones in one program language at a time and do not detect clones that span over multiple programming languages.

The accuracy of both emerging and proposed techniques need to be evaluated in detecting all types of clones. Several studies have been proposed to evaluate and compare clone detection tools [24, 156, 139, 125]. Three major techniques are used for evaluation: (1) manual inspection of reported clones to identify true positives and false positives. Usually, manual evaluation is needed to measure the accuracy of newly proposed techniques. (2) injecting the source code with artificially generated clones to measure how many clones the tool(s) can detect [153, 139], and 3) using benchmarks, with already identified and known clones in the system [101, 24, 153]. These three techniques work easily for syntactic clones. However, to apply these evaluation techniques for semantic clone detector, we face the following challenges: First, Semantic clones are not easy to detect [167]. Second, an ambiguity in the semantic clone definition makes it harder to manually evaluate the tools [168]. Third, there is no semantic clone benchmark available that helps to evaluate semantic clone tools. Existing benchmarks have a very limited number of semantic clone references.

In this thesis, we started with an extensive empirical study to investigate the effectiveness of applying different types of normalization, transformations and filtering in detecting semantic clones. In this study, we also investigated the developers' activity of copy-paste in open-source game systems. Then we performed a study of detecting functionally equivalent clones across .NET programming languages. We used the Common Intermediate Languages (CIL) that are generated by the Microsoft Visual Studio assembler. Inherently, we used the CIL to detect semantic clones for the .NET languages. We built a lightweight ontology to represent the CIL knowledge before we apply a selected matching algorithm. Our challenge in evaluating our technique was the lack of a real semantic clone benchmark. Therefore, we proposed a methodology to build a semantic clone benchmark (SemanticCloneBench). Besides, we defined a comprehensive taxonomy of semantic code clones that helps us to create semantic clones. Finally, we used our taxonomy and benchmark to evaluate and compare semantic code clone detectors.

1.1 Research Problem

Code cloning research is an active research area. A survey by Svajlenko [159, 152] reported more than 200 clone detection tools were available in 2017. These tools have been used in thousands of studies related to

code clones. Another survey by Mondal [124] reported 97 published papers on clone refactoring and tracking. Research in code clones is divided into four areas: clone analysis, clone detection, clone evaluation, and clone management. Figure 1.1 summarizes the cloning research areas. All clone research areas have been active, and a significant number of papers have been published in each area.

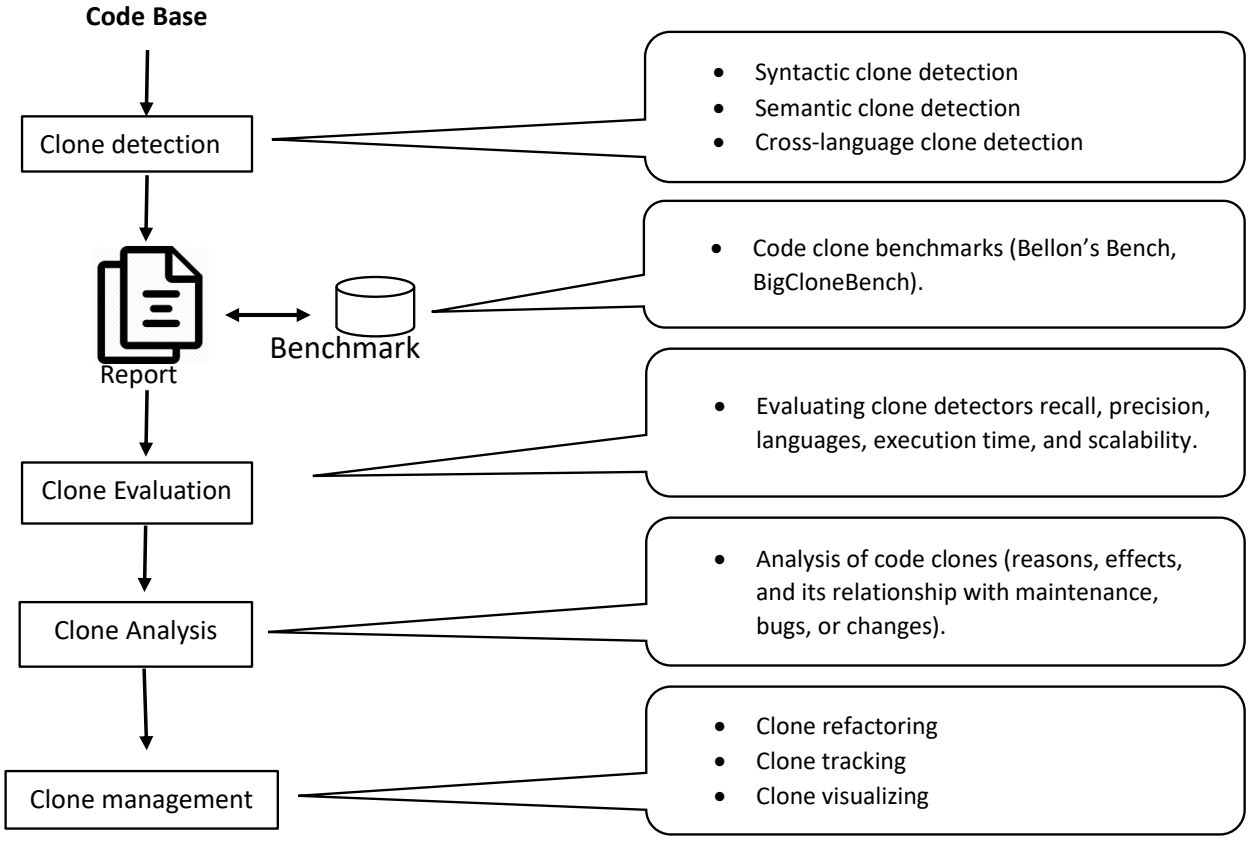


Figure 1.1: Code cloning research areas

Clone analysis involves empirical studies that analyze various clone aspects. Some of these studies focus on clone positive or negative effects [67, 86, 122]. Other research [141] tried to understand the reasons that make developers clone or avoid clone. More detailed studies [121, 177, 120] tried to investigate which type of clones are more harmful to maintenance, propagate bugs, or evolve inconsistently. Despite the number of empirical studies that analyze different aspects of code clones, there have been fewer studies that focus on these aspect for semantic clones. We believe that there is a need to analyze and understand the different traits of semantic clones in the source code, their reasons, effects, and importance.

In the code *clone detection* research area, practitioners propose tools and techniques to locate code clones in the code base. Practitioners compete to develop better tools to identify all possible clones in software systems efficiently and accurately. The clone detection process involves pre-processing the source code,

transforming it into an intermediate representation (string, token, tree, graph), and then applying matching to find candidate clones. Some used the byte and intermediate code to detect clones [25]. A great many studies [89, 69, 45] have been conducted to detect all types of clones. However, fewer studies have been performed to detect semantic clones [141, 38, 35, 107] and cross-language clones [11, 115]. Identifying equivalent functionalities in the system plays an important role in code comprehension, software maintenance and quality. Equivalent functionalities exist in software systems and are implemented in the same programming language (semantic clones) or in multiple programming languages (cross-language clones).

Despite the large number of proposed detection tools and techniques, there is a need for a fast and accurate semantic clone detection tool and cross-language clone detection tool. That is, there is a growing need for semantic clone detection techniques; an open problem in the area of clone detection. Clone evaluation studies indicate that there is a need for a more accurate semantic clone detection technique. A fast and accurate semantic clone detection tool will benefit cloning research and could be re-used in different cloning research areas, including semantic code clone analysis, refactoring and evaluation.

Clone detection *evaluation* involves the quantitative and qualitative evaluation of clone detection tools and techniques; it could be evaluating the accuracy of the tool and comparing it to other tools, or evaluating the performance and scalability of the tool, and the languages it supports. Tool evaluation gives practitioners a valuable guide for choosing the best tool for their application. Evaluating a tool's accuracy is accomplished by measuring the precision and recall of the tool. Measuring recall requires a benchmark. Very few clone benchmarks are available [24, 101, 153] and these benchmarks have very few semantic clones. Benchmarks have been used to evaluate both syntactic and semantic clone detection tools. Therefore, we believe there is a need for a semantic clone benchmark that can measure the accuracy of semantic clone detection tools. A semantic clone benchmark also could be used to measure the performance of syntactic clone detection tools in detecting semantic clones. As well, a semantic clone benchmark represents a good source of semantic clones that could be used for semantic clone analysis studies.

Code clone *management* consists of studies related to code clone tracking [43, 112, 147], refactoring [137, 124] and visualizing [118]. Managing code clones through refactoring and tracking offers many advantages to the developer to better understand the source code, make better design and implementation choices, and create better quality software. Also, clone management and tracking tools reduce maintenance efforts and reduce negative clone effects. Managing code clones is based on clone detection. Having better clone detection tools, semantic clone detection tools, and cross-language detection tools would support clone management tools and techniques.

Semantic clone-related research problems span over different code clone research areas (analysis, detection, evaluation, and management). In order to implement a semantic clone detection tool, I need to analyze and understand different types of semantic clones. As well, in order to evaluate the accuracy of the detection tool, I need a semantic clone benchmark. We can summarize the research problems related to semantic code clones as follows:

1. A need for an accurate and scalable semantic code clone detector.
2. A lack of a semantic clone benchmark for measuring clone detection recall.
3. A taxonomy of semantic clones that distinguishes differnt semantic clones and how they can be created.
4. A lack of empirical studies that evaluate both syntactic and semantic clone detection tools in detecting semantic clones.

1.2 Addressing the Research Problems

To address the problem of semantic clone detection, benchmarking and evaluation, I divided the research into sub-problems and carried out research studies for each one.

- *Code cloning in games and source code normalization toward semantic code clone detection: An empirical study*

  This study is divided into two main empirical studies about code cloning. The first study explores the status of code cloning in game software, compares it to non-game software, and investigates cloning across games. It answers a number of research questions. Do game software developers clone code as much as other software developers? are games considered to be of higher quality than other software in terms of code cloning? Do game developers have the intent to clone code from other games? To answer the research questions, we used 32 games and 9 non-games systems. The second part of this study addresses the effect of code transformations (normalization and filtering) on code clone detection. We applied eight different code transformations to the source code. Then we analyzed the quality and quantity of reported clones. This study answers the research questions, How does code transformation improve the detection process? Does applying code transformation help in detecting semantic clones? In this study, I analyzed in depth which code transformations enabled the detection of extra near-miss clones and semantic clones. Since Microsoft .NET produces a unified intermediate language for all of its programming languages, I used the intermediate language to detect functional clones (semantic clones and cross-language clones) and I applied multiple code filtering and transformation techniques to enhance the accuracy of detecting both semantic and cross-language clones.

- *Detecting clones across Microsoft .NET programming languages.*

  In this study, we present a clone detection approach for the .NET language family. The approach is based on the Common Intermediate Language, which is generated by the .NET compiler for the different languages within the .NET framework. In order to achieve an acceptable recall while maintaining the precision of our detection approach, we defined a set of filtering processes to reduce noise in the raw data. We have shown that these filters are essential for Intermediate Language-based clone detection without significantly affecting the precision of the detection approach. Finally, we studied the quantitative and

qualitative performance aspects of our clone detection approach. We evaluated the number of reported candidate clone-pairs, as well as the precision and recall (using manual validation) for several open source cross-language systems, to show the effectiveness of our proposed approach.

- *Semantic clone detection using intermediate language knowledge base.*

  Similar to the previous study, we used the .NET intermediate language to detect semantic clones. We built a byte code ontology that describes the intermediate code, its structure and relationship. Then we used this ontology to detect clones at the byte code level by using an ontology matching technique. After evaluating our technique and comparing it to state-of-the-art clone detectors (NiCad, ConQAT, and SimCad) [138, 41], we found that this technique can detect all types of clones (Type-1, Type-2, Type-3, and Type-4) with acceptable accuracy. Furthermore, it could detect more gapped clones and semantic clones that are not detectable by state-of-art clone detectors.

- *Semantic code clone benchmark.*

  Due to the need of having a comprehensive evaluation of our technique in detecting semantic clones, we find it is important to build a semantic code clone benchmark. This benchmark could be used to evaluate existing semantic clone detectors as well as new techniques and tools. The main limitations of available benchmarks include: they are restricted to one programming language; they have a limited number of clone pairs that are confined within the selected system(s); they require manual validation. To overcome these limitations, we proposed a methodology to generate a wide range of clone benchmarks (SemanticCloneBench) for different programming languages with minimal human validation. Our technique is based on the knowledge provided by developers who participate in the crowd-sourced information website, Stack Overflow. The idea is based on mining the answers that solve the same programming question, and implemented using different syntax.

- *Evaluation of semantic code clone detection tools.*

  In this study, we used our benchmark to evaluate and compare a selected semantic clone detection tools by measuring recall, precision and execution time. We implemented an injection and evaluation framework that measures the recall for detection tools automatically. The framework takes clones from a given benchmark, injects them into a selected subject system, runs the tool under study, and evaluates the tool's result by measure the recall automatically. Then we build another semantic clone benchmark manually that follows the definition of semantic clones. We started by reviewing the literature to classify all possible types of semantic clones. Then we defined taxonomy for all possible semantic clones that we used to build the benchmark. Similarly, we used this benchmark and the injection framework to evaluate the selected tools.

1.3 Outline of the Thesis

This thesis is organized into the following chapters:

- **Chapter 1** introduces the research problems related to semantic code clones and a short description of the studies performed to address them.
- **Chapter 2** talks about the basic concepts related to semantic clones, which includes the definition of code clones, code clone detection, benchmarking and evaluation of detection tools.
- **Chapter 3** presents our empirical study of code cloning in game software systems. Also, it discusses the effect of clone transformation in clone detection.
- **Chapter 4** presents our study in clone detection across Microsoft .NET programming languages.
- **Chapter 5** describes our technique to detect semantic code clones using the .NET intermediate language.
- **Chapter 6** shows our process in building a semantic clone benchmark, SemanticCloneBench.
- **Chapter 7** contains our study of using the SemanticCloneBench to evaluate semantic clone detection tools. In this study, we defined a semantic code clone taxonomy, and used it to create another benchmark that is used to evaluate the same detectors.
- **Chapter 8** concludes the thesis and discusses future work.

1.4 Manuscript-Style Thesis

This thesis is written in the manuscript style. Each main chapter in the thesis has been written according to published or unpublished paper to an academic conference or journal. Each manuscript is edited and formatted to fit in the thesis. For all the included work, I was the main author and the lead researcher who did all the experiments and wrote the manuscripts. Co-authors of manuscripts have the supervisory role. At the beginning of each chapter, I referred to the manuscript, and the contribution of any co-authors. The following is the list of the manuscripts used in the thesis:

- Farouq Al-omari, I. Keivanloo, C. K. Roy, and J. Rilling. Detecting Clones Across Microsoft .NET Programming Languages. In 2012 19th Working Conference on Reverse Engineering, pages 405–414, 2012.(Chapter 4).
- Farouq Al-omari and C. K. Roy. Is Code Cloning in Games Really Different? In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16), page 1512–1519. Association for Computing Machinery, 2016.(Chapter 3).

- Farouq Al-omari, C. K. Roy, and T. Chen. SemanticCloneBench: A Semantic Code Clone Benchmark Using Crowd-Source Knowledge. In 2020 IEEE 14th International Workshop on Software Clones (IWSC), pages 57–63, 2020.(Chapter 6).

- Farouq Al-omari and C. K. Roy. Toward Semantic Code Clone Detection and Evaluation: The Capability of Microsoft .NET Intermediate language in Detecting Semantic Clones. Science of Computer Programming, 18 pp., 2020 (submitted). (Chapter 5).

- Farouq Al-omari and C. K. Roy. Comparing and Evaluating Semantic Code Clone Detectors (Unpublished). (Chapter 7).

Chapter 2

Background

In this chapter, we discuss the basic terminologies related to code clones. This chapter includes the definition of code clones, types of code clones, a brief review of code clone detection, finally, a review of clone benchmarking.

2.1 Code Clone Definition and Types

A code fragment in the source code that is identical or similar to another code fragment in the code base is considered a code clone to the second and both called a clone pair. This definition is based on the concept of similarity. The similarity could be textual, syntactic or semantic. In the literature, the following definition of clone has been widely acceptable [140]:

Type-1: Identical code fragments except for variations in whitespace (may also have variations in layout) and comments.

Type-2: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.

Type-3: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

Type-4: Two or more code fragments that perform the same computation but are implemented through different syntactic variants.

Type-1, Type-2, and Type-3 clones are based on textual similarity. A code's fragments are considered clones if they are textually similar, even if they are functionally different. Textual clones are more common in software code base because they are usually the result of copy/paste practice. Conversely, Semantic clones hold no textual similarity.

2.1.1 Type-1 Clones

Type-1 clones are usually resulted by copy and paste without modifications or just modifying the layout and the comments. Figure 2.1 shows an example of Type-1 clone. Type-1 clones are easy to detect by most clone detectors.

```java
static void bubbleSort(int[] arr) {
     int n = arr.length;
     int temp = 0;
     for(int i=0; i < n; i++){
         for(int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j]){
               //swap elements
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp;
             }
         }
     }
 }
```

```java
static void bubbleSort(int[] arr) {
     int n = arr.length;
     int temp = 0;
     for(int i=0; i < n; i++){
         for(int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j])
             {
               //swap temp with arr[j-1]
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp; }
         }
     }
 }
```

Figure 2.1: Type-1 clone example

2.1.2 Type-2 Clones

Similar to Type-1 clones, Type-2 clones are usually created by copy and paste followed by a small modification such as renaming a variable, an argument or a literal. Type-2 clones are easy to detect using most detectors. As most detectors use a similarity threshold of 70% up to 100%, they can detect Type-2 clones. Figure 2.2 shows an example of a Type-2 clone.

```java
static void bubbleSort(int[] arr) {
     int n = arr.length;
     int temp = 0;
     for(int i=0; i < n; i++){
         for(int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j]){
               //swap elements
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp;
             }
         }
     }
 }
```

```java
static void bubbleSort(int[] arr) {
     int len = arr.length;
     int temp = 0;
     for(int i=0; i < len; i++){
         for(int j=1; j < (len-i); j++){
            if(arr[j-1] > arr[j]){
               //swap elements
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp;
             }
         }
     }
 }
```

Figure 2.2: Type-2 clone example

2.1.3 Type-3 Clones

Making more modifications to a copied code fragment at the level of a line, such as adding, deleting or modifying a line or more, will result in a Type-3 clone. Figure 2.3 shows an example of a Type-3 clone. Type-3 clones could result from copy, paste and modification. Also, they could be accidentally resulted when developers solve the same problem and use the same programming style.

```
static void bubbleSort(int[] arr) {
     int n = arr.length;
     int temp = 0;
     for(int i=0; i < n; i++){
         for(int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j]){
               //swap elements
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp;
             }
         }
     }
 }
```

```
static void bubbleSort(int[] inputArr) {
     int len = inputArr.length;
     for(int i=0; i < len; i++){
         for(int j=1; j < (len-i); j++){
            if(inputArr[j-1] > inputArr[j]){
               //swap elements
               int temp = inputArr [j-1];
               inputArr[j-1] = inputArr[j];
               inputArr[j] = temp;
             }
         }
     }
 }
```

Figure 2.3: Type-3 clone example

2.1.4 Type-4 Clones

Type-4 clones (some times referred to as functional clones or dependence clones) are the results of semantic similarity between two or more code fragments. In this type of clone, the cloned fragment does not share the original syntax or structure. Two different programmers could develop two code fragments to do the same task, making the code fragments same in their functionality yet different syntactically. Semantic clones are challenging to detect since they could be implemented with different syntax and a single change in a code fragment could change the meaning (functionality) of the fragment. The best example for semantic clones are sorting algorithms (merge sort, quick sort, and bubble sort). They all do the same functionality and are implemented in different ways, syntactically, and structurally (Figure 2.4).

```
static void bubbleSort(int[] arr) {
     int n = arr.length;
     int temp = 0;
     for(int i=0; i < n; i++){
         for(int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j]){
               //swap elements
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp;
             }
         }
     }
 }
```

```
public static void selectionSort(int arr[]) {
       int n = arr.length;
       for (int i = 0; i < n-1; i++)
       {
              int min_idx = i;
              for (int j = i+1; j < n; j++)
                     if (arr[j] < arr[min_idx])
                            min_idx = j;
              // swap the min with first
              int temp = arr[min_idx];
              arr[min_idx] = arr[i];
              arr[i] = temp;
       }
}
```

Figure 2.4: Type-4 clone example

Some practitioners consider syntactic clones that have similar functionality to be semantic clones while others define semantic clones as functionally similar and syntactically different. [141] defines semantic clones as functionally similar clones and implemented using a different syntactic; while [39, 46, 47] consider functionally similar code fragments as semantic clones regardless of its syntactic.

The terms relative clones [149], redundant code [105, 150], dependent clones [38], functional clones [75],

functionally similar clones [50, 161, 103] and Type-4 clones [141] are widely used to refer to semantic clones. A number of definitions have been proposed in the literature for semantic clones. Some of them narrow the definition of semantic clone to one type of semantic while others used a wider (non-specific) meaning of semantic. The following is a summary of different definitions that have been proposed:

- Gabel et al. [53] defined semantic clone as a pair of code fragments that have similar control flow or the same program dependence graph. This definition does not distinguish between syntax clones and semantic clones, since Type-1 clones and some of Type-2 and Type-3 clones have the same control flow or same program dependence graph. Furthermore, not all semantic clones have the same control flow or same program dependence graph.

- Simion: The term Simion was first used by Juergens et al. [77] to refer to a similar I/O behaviour code. A number of studies [161, 149, 150, 130] considered methods that produce same output results using the same input set, semantic clones. Simons represent all clones that produce the same output. Similar to Gabel's definition, this definition does not distinguish between syntax clones and semantic clones. Also, not all semantic clones produce the same output.

- Semantically similar clone: Two code fragments are considered semantically similar clones if they have similar control flow and have many overlapping statements [173]. Also, Elva and Leavens [46] used the same name, and they defined it as code fragments that perform the same function most of the time. These definitions of semantic clone still need more clarification with examples to show what semantically similar means.

- Functionally similar code (FSC): Are clones that have the same or similar functionality but are created independently or implemented differently [103, 167]. Tajima et al. [161] defined FSC as different code that have similar functionality. Functional similar clone definition refers to all clone types that have the same functionality regardless of their syntax similarity.

- Keivanloo et al. [87] defined two methods as a clone if they are either similar in their patterns or functionalities. This is a wide definition that refers to coding patterns or functionalities.

- As noted above, Type-4 clones by Roy et al. [140] is the most widely accepted definition for semantic clones.

In summary, all mentioned definitions agree that semantic clones have the same functionality. However, the definitions that express similar functionality are still vague since it is not clear which changes could be applied to a code fragment and could preserve the functionality, align it or change it totally. Most of the definitions agreed that semantic clones do not have syntactic similarity since they are classified as semantic clone. More examples are needed to show the types of semantic transformation in the source code and how the semantics are represented in different syntactic constructs, and how a small syntactic change can change the semantic.

2.1.5 Cloning Related Terms

Different cloning terms and terminologies are used in clone documentation and reports. The following clone terminologies are used in this thesis.

- **Code fragment** is a contiguous piece of source code within a source file. Code fragment specified by file name, start line, and end line.
- **Clone pairs** are two code fragments that are similar to each other.
- **Clone class** is a set of code fragments that are similar to each other.
- **Exact clones** are two identical code fragments. Exact clones are Type-1 clones.
- **Near-miss clones** refer to Type-2 and Type-3 close. Near-miss clones are not exact clones but have the same syntax. Usually, near-miss clones are created by applying some edits to one code fragment, such as identifies renaming.
- **Gapped clones** are code clones that are created by some modification. Gapped clones refer to Type-2 and Type-3 clones.

2.2 Code Clone Detection

In code clone detection, tools are developed to identify the location of clone fragments in the codebase. We found there are hundreds of proposed techniques to identify reused code (cloned code). Each technique targets one or more clone types (Type-1, Type-2, Type-3, and Type-4). The detection process consists of a series of steps that includes:

- Parsing the source code to extract the target code fragments at desired granularities (block, method, class or file).
- The extracted code fragments are transformed into intermediate representation for comparison purpose. Code transformation includes pretty-printing, removing of comments or whitespaces, tokenization, generating an abstract syntax tree (AST) or generating a program dependence graph.
- Comparing the processed units to identify similar code fragments. Different algorithms used depending on the representation of code (text, tokens, tree or graph).
- Finally, the detection tools report detected clones as clone pairs or clone classes. Each clone pair is defined by a pair of code fragments, where each code fragment is specified by the file name, the start line and the end line of the code fragment. However, some tools identify fragments by file name and method name.

A large number of clone detection techniques are proposed in the literature, and new techniques are still emerging, as outlined in surveys by Roy et al. [141] and Ain et al. [9]. The majority of detection tools can detect one type or two types accurately. For example, NiCad [138], CloneWorks [158], SourccerCC [144], can detect clones Type-1, Type-2 and Type-3 with high precision and recall. But, Type-4 clone detection is still a big challenge in code clone research. However, there are a number of studies and tools have been proposed to detect Type-4 clones [65, 87, 47, 53]. But, most of these techniques suffer from two major challenges in semantic clone detection. First, detecting semantic clone with good accuracy. Second, detecting semantic clone fast enough. The majority of semantic clone detectors use the PGD to detect clones and sub-graph match is very expensive.

2.3 Evaluation of Clone Detection

With the large number of available and newly emerging clone detection tools and techniques, there is an ongoing need to evaluate and compare these tools. The selection of a clone detection tool for a specific purpose is based on different factors, such as the accuracy of the tool, the language it supports, the performance and scalability of the tool. All of these factors are publicly available or easy to measure except the accuracy. Even though tool's accuracy is the key factor for selecting tools. The two information retrieval accuracy measurements, recall and precision are used to measure the accuracy of detection tools.

Figure 2.5 illustrates the accuracy measurements, precision and recall. The highlighted left area (in gray) represents actual clones in a system, while the right side represents non-cloned code in the system. Let us consider the set of clones inside the blue rounded rectangle represents clones reported by a tool. Then clones in the system are divided into four regions that are labelled by, false negative, true positive, false positive and true negative.

True positive (TP) represents the set of true clones that are successfully detected by the tool.

False positive (FN) represents the set of non-cloned fragments that are reported as true clones by the tool.

False negative (FN) represent the set of true clones that the tool failed to detect.

True negative (TN) are the set of non-cloned fragments that the tool also does not report as clones.

Precision is defined as the number of true clones detected divided by the total number of detected clones; look at Equation 2.1, where TP is the number of true positives and FP is the number of false positives. Tools should report more true positives and less false positives. Precision reflects the accuracy of reported clones. To measure the precision of a tool, reported clones need to be validated. Validating all reported clones needs a lot of effort when it is done manually. Therefore, practitioners validate a random sample of the report.

$$Precision = \frac{TP}{TP + FP} \tag{2.1}$$

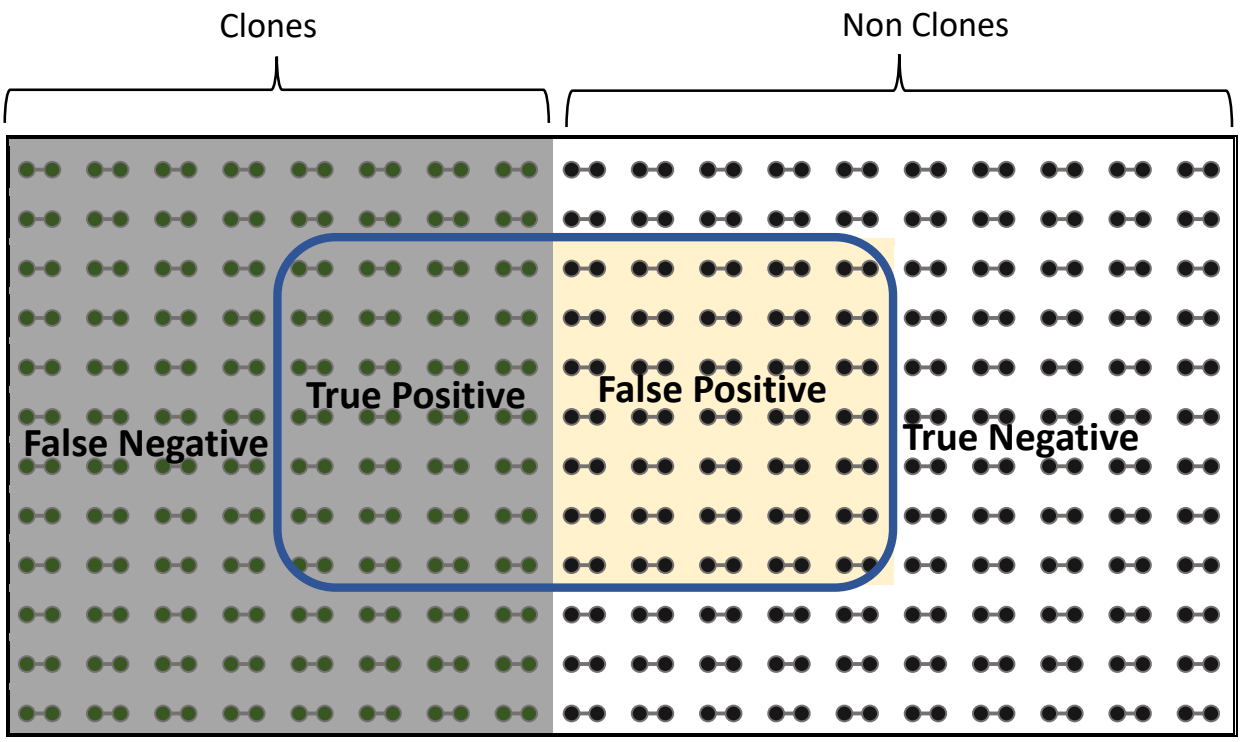


Figure 2.5: Recall and precision

Recall is defined as the number of true clones detected divided by the total number of true clones system. Equation 2.2 defines recall, where TP is the number of true positives and FN is the number of false negatives. Recall measures the ability of tools to detect clones that exist in the system. To measure recall of a tool, all true clones in the system should be known.

$$Recall = \frac{TP}{TP + FN} \tag{2.2}$$

2.3.1 Code Clone Benchmark

A code clone benchmark could be defined as a software system in which all code fragment pairs are tagged as either true clones or non-cloned code. The existence of benchmarks enables us to measure the actual precision and recall. However, manual identification of all clones and non-clones in a system requires extensive time and efforts i.e. a software system of 2000 methods requires a manual validation of $n * (n - 1)/2$ 2000 X 2000, (1999000 unique method pairs). To overcome this challenge of extensive manual validation practitioner suggested the following alternatives:

- Manually validate a random set of the total method pairs in a system, and consider this set as the benchmark.

- Manually validate a random set of the total clones detected by a tool or tools in a system, and consider this set as the benchmark.
- Manually or automatic creation of clones and inject them into a system, and consider the injected set as the benchmark.

Chapter 3

Code Cloning in Games and Source Code Normalization Toward Semantic Code Clone Detection: An empirical Study

In this chapter, we discuss a comprehensive empirical study in code cloning. Our research focuses on two aspects of code cloning: first, code cloning status in open source game systems. The second aspect is code transformation and its effect on clone detection.

Since there are a tremendous number of similar functionalities related to images, 3D graphics, sounds, and script in games software, there is a common wisdom that there might be more cloned code in games compared to traditional software. Also, there might be more cloned code across games since many of these games share similar strategies and libraries. In this study, we attempt to investigate whether such statements are true by conducting a large-scale empirical study using 32 games and 9 non-games software systems, written in three different programming languages, C, Java, and C#, for the case of both exact and near-miss clones. Using a hybrid clone detection tool NiCad [138], and a visualization tool VisCad [17], we examine and compare the cloning status in games and compare it to the non-games, and examine the cloned methods across game engines.

Code clone detection tools try to detect clones with higher accuracy, i.e. detect more true clones and avoid as false positives as far possible. Most detection tools apply some kinds of code transformation before the comparison (detection) to achieve better results [141]. Some tools apply text processing and transformation [108, 144, 170], and other tools transform the source code into other representations such as AST or PDG [107, 49, 56, 178]. However, some tools use the byte code or intermediate code [25]. NiCad, like all other tools, applies a wide range of transformations to the source code. NiCad transformation includes code filtering, a set of abstractions and normalization. In this chapter, we discover how these transformations improve clone detection? And if we combine some transformations, is it possible to detect some semantic code clones?

This chapter is an updated and extended version of our manuscript [13] "Is code cloning in games really different?" Which was published in the 31st ACM/SIGAPP Symposium on Applied Computing(SAC 2016). I was the lead author of the paper under the supervision of Chanchal K. Roy.

This chapter is organized as follows. Section 3.1 introduces the chapter by discussing the research questions. Section 3.2 describes the code normalization provided by NiCad. In Section 3.3, we setup our experi-

ment. We discuss the experimental results and analysis in Section 3.5. Related work is presented in Section 3.6. Section 3.7 discusses possible threats to validity of our study, followed by the summary in Section 3.8.

3.1 Introduction

Code cloning or code reusing is an inevitable practice by programmers [141]. Programmers usually copy and paste their code. In some cases, programmers look through the Internet or related open source systems for a tested and ready to use source code [137]. Programmers reuse the source code of open source systems not only for the low cost and ease of access, but also possibly for its quality[137, 86]. Large software systems contain 10-15% of duplicated code, which are considered as code clones [86].

Fowler and Beck [52] consider duplicated code as the #1 code bad smell. Furthermore, they describe a number of ways to remove clones to make programs easier to understand and maintain. Empirical studies [133, 91] reported that while code cloning is a principle reengineering technique and could be useful in many ways [86, 163], they are harmful to software and degrade software quality [44, 83, 23]. For example, if a code fragment contains some form of defects; all copies of that fragment should be located and fixed [141].

Open source systems are popular not only because of free access, but also their quality. Apache and Linux are good examples of open source systems both in terms of quality and popularity. Not only open community can leverage the benefits of open source systems, but also commercial software vendors exploit such invaluable resources [117]. Mockus [117] indicates that there is a large scale reuse of code between open source software that is represented in three forms: entire files, functions, and templates. Even core developers have the intention to reuse code across their projects at all stages of the development process. Also, they use tools to search and integrate the reused code fragments [61]. Therefore, the reused source code across open source systems in each domain needs to be analyzed. However, an empirical study could not (possibly should not) cover all open source domains. Therefore, in this study, we aim to conduct an empirical study for a growing open source domain, the games.

Usually, game software contains common functionalities related to 2D images, 3D models with animation, collision maps, sound, physics, and artificial inelegance. Also, most games are built upon application programming interface (API) such as OpenGL and libraries such as DirectX. Such commonality suggests that,

Hypothesis 1: *There may be more copy/paste clones in games than other software systems.*

In games industry, even whole games may need to be cloned. For example, creating a new first-person shooter game would be easier by cloning the idea of all previous first-person shooter games. Usually, game development has more challenges than development of other application-oriented noncritical software systems. Games need to be feature complete, unlike other software that could be released with missing features. Most games require a high level of graphic design and other challenges that pose programmers to clone some components from existing ones. Developers can copyright part of their project as maps or characters.

However, it is not possible to protect the idea, software design, or game mechanics from cloning, in particular in the open source community. Often open source games are recreated (cloned) to add new features, support new hardware, or support of a different platform or operating system. There is thus a conventional wisdom that,

Hypothesis 2: *There may be more cross games copy/paste clones in open source games than other software systems.*

However, to our knowledge none of the hypotheses above were validated with empirical evidence. Therefore, in this empirical study, we aim to examine how much copy/paste clones are in game software and whether it is different from traditionl software. We further analyze in detail the status of shared code between games and whether software clones occur between different game genres, same game genres, and cloned (recreated) games. Game genres is a categorization of games according to the way player interacts with a game. Examples of game genre are action games, shooting games, strategy games, and others.

By reviewing the definition of clone types, we can see that applying some kind of filtering, normalization, or transformation to clones type-2 converts them to clones Type-1. Also, applying some filtering, normalization, and/or transformation to clones Type-3 could convert them to clones Type-2 or clones Type-1. NiCad has been proven to detect Type-1, Type-2 and Type-3 with high precision and recall. Also, NiCad is empowered with some code transformation options. In this chapter, we are trying to discover what type of extra clones do these code transformations enable us to detect and do applying these code transformations enables us to detect semantic clones?

In this study, we thus conduct an empirical study with 32 games of three different languages C, Java, and C# using one of the state-of-art clone detector, NiCad [138] and a visualization and analysis system, VisCad [17]. Furthermore, in order to compare the findings with non-game open source software, we use nine non-game software as of our previous study [136]. Also, we used NiCad code transformation options to analyze what type of clones these transformations enable us to detect. Our study aims to answer the following research questions:

RQ1: What is the copy/pasted cloning (reusing either with direct copy or with modifications) status in open source games for different programming languages?
RQ2: How does such level of reusing of code fragments differ between games and non-game software systems? As of **Hypothesis 1**, are there more clones in games than non-games open source systems?
RQ3: How often developers reuse functionalities from other related games? As of **Hypothesis 2**, is there any evidence of copy/paste reuse among different games, either related or different?
RQ4: *How does code transformation improve the detection process accuracy?*
RQ3: *Does applying code transformation help in detecting semantic clones?*

Our experimental results show that while there is indeed some evidence of copy/paste reuse in open source games, the extent of reuse is not that significant as compared to non-games and that there is little evidence of cross-game copy/paste reuse.

3.2 Source Code Transformation

Almost all detection tools [11, 141, 144, 154] do a series of prepossessing, transformation, and re-representation of the source code or byte code -in some cases- before matching (identifying potential clones). The goals of the mentioned steps are done in order to:

1. Eliminate unnecessary data for the detection process and keep meaningful data for the detection purpose.
2. Reduce processed data and reduce processing time for the comparison (matching algorithm).
3. Detect more potential clones (increase true positive clones).
4. Avoid noise that could result in false positive clones (reduce false-positive clones).

The second common practice by most detection tools is the selection of a threshold. Most detection tools use a matching algorithm such as Levenshtein distance, Longest Common Subsequence (LCS), or Jaccard ratio. The more relaxation on the selected threshold, the more matches results and more clones detected. Most tools set their threshold value before (or when) they start to produce false-positive clones.

3.2.1 Code Transformation

Detection tools clean up the source code, filter unnecessary data, or reformat and re-represent it before detection is applied. The ultimate goal is to detect as many true clones and avoid false clones. There is no empirical study so far that analyses the effect of code transformation on the detection process and especially on semantic code detection. In this chapter, we selected the clone detector, NiCad [141], that support different forms of code transformation. Then we analyze the contribution of each level of processing in clone detection. Finally, we evaluate the resulted clones. The following are some of the code preprocessing and transformations provided by NiCad.

Pretty printing

NiCad Uses TXL parser to parse the source code and extract the desired code fragments. It pretty-prints the extracted code into a consistent layout. Pretty-printed code eliminates the differences that resulted from the code formatting, style, or layout. Listing 3.1 shows a pretty-printed Java method. Also, TXL parser ignores comments in the source code.

Filtering

Code filtering means to remove some syntax elements in the source code that can cause a mismatch of similar code fragments or produce false positives. Besides its contribution to detection results, filtering reduces processing time and hardware requirements. All detection tools filter out comments and white spaces. NiCad processor has the option to filter out declaration statements. Listeng 3.2 shows the same method in Listing 3.1 after filtering out the declaration statements.

Abstraction

Abstraction is the replacement of selected code syntax into a more general representation. NiCad provides some abstractions, abstraction of code blocks, abstraction of declarations, abstraction of literals, abstraction of conditions, and abstraction of expressions. Listings 3.4, 3.3, 3.4, 3.5, 3.6 and 3.7 show examples of each type of abstraction.

Code normalization

In code normalization, some units in source code are transformed into a certain normal form. For example, identifiers in Listing 3.1 could be consistently renamed, as shown in Listing 3.8 or arbitrarily renamed as shown in Listing 3.9.

Listing 3.1: Java method before any processing

```
1  public static Record stat (int [] x) {
2  int max = x [0];
3  int sum = 0;
4  for (int i = 0;
5  i < x.length; i ++) {
6  if (x [i] >= max) max = x [i];
7  sum = sum + x [i];
8  }
9  Record rec = new Record (max, sum);
10 return rec;
11 }
```

Listing 3.2: Java method after filtering declaration

```
1  public static Record stat (int [] x) {
2  for (int i = 0;
3  i < x.length; i ++) {
4  if (x [i] >= max) max = x [i];
5  sum = sum + x [i];
6  }
7  return rec;
8  }
```

Listing 3.3: Java method after abstraction of block

```
1  public static Record stat (int [] x) {
2  int max = x [0];
3  int sum = 0;
4  for (int i = 0;
5  i < x.length; i ++) block
6  Record rec = new Record (max, sum);
7  return rec;
8  }
```

Listing 3.4: Java method after abstraction of declaration

```
public static Record stat (int [] x) {
declaration declaration for (int i = 0;
i < x.length; i ++) {
if (x [i] >= max) max = x [i];

sum = sum + x [i];
}
declaration return rec;
}
```

Listing 3.5: Java method after abstraction of lireral

```
public static Record stat (int [] x) {
int max = x [literal];
int sum = literal;
for (int i = literal;
i < x.length; i ++) {
if (x [i] >= max) max = x [i];

sum = sum + x [i];
}
Record rec = new Record (max, sum);
return rec;
}
```

Listing 3.6: Java method after abstraction of condition

```
public static Record stat (int [] x) {
int max = x [0];
int sum = 0;
for (int i = 0;
i < x.length; i ++) {
if (condition) max = x [i];

sum = sum + x [i];
}
Record rec = new Record (max, sum);
return rec;
}
```

Listing 3.7: Java method after abstraction of expression

```
public static Record stat (int [] x) {
int max = expression;
int sum = expression;
for (int i = expression;
expression; expression) {
if (expression) expression;

expression;
}
Record rec = expression;
return expression;
}
```

Listing 3.8: Java method after normalize all identifiers

```
public static x1 x2 (int [] x3) {
int x4 = x3 [0];
int x5 = 0;
for (int x6 = 0;
x6 < x3.x7; x6 ++) {
if (x3 [x6] >= x4) x4 = x3 [x6];

x5 = x5 + x3 [x6];
}
x1 x8 = new x1 (x4, x5);
return x8;
}
```

Listing 3.9: Java method after normalize all identifiers

```
public static x x (int [] x) {
  int x = x [0];
  int x = 0;
  for (int x = 0;
  x < x.x; x ++) {
    if (x [x] >= x) x = x [x];

    x = x + x [x];
  }
  x x = new x (x, x);
  return x;
}
```

3.3 Experimental Setup

Subject Systems: In this study, we analyzed 32 open source games for three different programming languages C, C#, and Java. Table 3.1 summarizes these systems. For each programming language, we choose a number of open source games from different categories (e.g., shooting, strategy, and racing) and with different sizes. In addition, we compared the cloning/reusability status of the selected games open source to another group of a non-game open source. Therefore, we selected another group of non-game systems for this section of our study. Roy and Cordy [136] studied the clone status in these systems (i.e., non-games). Table 3.2 summarizes these subject systems.

Tool Settings: For the purpose of reusability/clone detection, we used NiCad [138], a state-of-art clone detection tool. NiCad is a scalable and flexible clone detection tool with high recall and precision in detecting identical and near-miss intentional clones [136].

In this study, however, we did not aim at studying different types of clones. Rather, we aimed to study the extent of direct copy/paste with or without minor adaptations, which we call intentional clones [136]. We apply a standard pretty-printing using NiCad that removes the comments and makes the code consistent in terms of whitespaces and formatting. After that, we used different similarity thresholds of NiCad (UPI thresholds). In particular, we used UPI thresholds of 0.0, 0.1, 0.2 and 0.3. When we use UPI threshold 0.0 on the pretty-printed code, we can find out the exact copy/pasted code. When UPI threshold 0.1 is used, we find almost similar code where there could be only 10% differences (i.e., 90% similar) between the cloned/reused fragments. Similarly, when UPI thresholds 0.2 and 0.3 have been used to determine the extent of similarity (20% and 30% differences, respectively) and reuse of source code in the subject systems. Again, the benefits of using pretty-printing and these thresholds is that they help us find the intentional copy/paste/reuse code fragments instead of just similar fragments.

In order to analyze the detected clones, we used VisCad [17], a clone visualization tool that supports a number of clone detectors. In addition to visualization, VisCad offers a number of clone related metrics that facilitate analysis of clones.

Table 3.1: Selected game subject systems in the study

Language	Subject system	NOF	LOC	NOM
Java	Freecol	689	193k	8472
	Robocode	653	94k	5502
	AndEngine	596	66k	4621
	Greenfoot	345	60k	2907
	Terasology	614	86k	5754
	Env3D	75	8k	741
	Jake2	258	85k	4160
C#	Mono Game	791	166k	3587
	Unity Steer	51	515k	115
	OpenRA	630	643k	2121
	Samurai	82	568k	228
	Sleep Walker	528	616k	2010
	NetGore	603	66k	2016
	Axiom Engine	421	121k	4273
C	Freedroid	128	77k	2414
	Berlios	188	162k	2601
	Quake 3	428	347k	7798
	Open Arena	171	147k	3232
	Egoboo	179	194k	1636
	Nethack	254	222k	2100
	Chocolate Doom	174	103k	1792
	FreeCiv	459	285k	11330
	SGE	201	44k	342
	Hexan	56	70k	1224
	Wolf3D	28	15k	558
	Chocolate Doom	159	86k	1771
	Quake 2	181	149k	4472
	Enemy Territory	456	473k	9096
	QW	112	78k	1780
	WinQuake	127	88k	2082
	Quake 2	181	149k	4477
	Quake 3	428	348k	7798

Table 3.2: Non-games subject systems

Lang	Subject system	NOF	LOC	NOM
Java	Eclipse- jdt core	1202	455k	18257
	CIRC	65	127k	765
	JHotDraw	469	189k	2886
C#	Mono c# compiler	54	880k	1993
	Castle	2407	270k	9236
	Nant-0.68	429	104k	2335
C	Apache http	252	200k	3878
	Postgresql	1155	1018k	15399
	Cook	296	73987	1335

3.4 Clone Metrics

Total Cloned Method (TCM): It is common to measure clone density in a system by the number of clone pairs and clone classes. However, in this metric, we are using the total number of cloned methods in the system to represent clone density. A method is considered cloned method if it forms at least a clone pair with another method in the system. We also used Total Cloned Method Percentage (TCMp), the percent of cloned methods in the system.

Total Cloned Line of Code (TCLOC): Since some cloned methods are small and others are large in size, we considered the cloned method size (Lines of Code, LOC) as a metric for our study. A Total Cloned Line of Code (TCLOC) metric is used to determine the total cloned line of code in the system. Another inherited metric also used *TCLOCp* that represents the percentage of cloned line of code in the system.

File Associated with clones (FAWC): The previous metrics measure the number and percentage of cloned methods and their size. However, it is still important to know the portion of files that hold these clones and how/where these clones are distributed in the system. Therefore, FAWC reports the number of files that contain at least a cloned method. Also, *FAWCp* is defined as the percentage of files associated with clones in the system.

Clone Class Radius (CCR): Previous metrics show the density of clones in the subject system. We still need to measure the distribution of these clones in the system. Thus by clone class radius, we mean the average distance between cloned files in clone classes and their lowest common ancestor directory in the file system. The larger CCR, the more scattered cloned file class in the system; therefore, the more maintenance effort. Also, we refer to Clone Class Size as CCS, which is the number of cloned fragment in the class. Other code metrics that we used: (1) LOC: Lines of Code, (2) NOF: Number of Files, (3) NOM: Number of Methods, (4) NOCC: Number of Clone Classes, (5) NOCP: Number of Clone Pairs.

3.5 Experiment Results

In this section, we discuss the experiment results of the study. Each subsection of the experiment is designed to answer a research question. In the discussion, we provide the overall findings and describe the statistical measures. The following subsections describe in detail our results and findings.

3.5.1 Cloning Status in Game Open Source

In this subsection, we answer ***RQ1*** by reporting our findings regarding clone status in open source games developed using three different programming languages. We carried out extensive analysis on 32 systems using the selected metrics (Section 3) to have a deeper insight into the clone status in open source games.

3.5.1.1 Overall Cloning Density

In this subsection, we provide the overall cloning level for different games open source systems in different programming languages. Although there is debate about the effect of clones in software, clones are likely harmful to the system, and they increase the effort of maintenance and refactoring [119, 111]. We used both TCMp and TCLOCp metrics as a measure of cloning level. Figure 3.1 shows the percentage of cloned functions for each programming language system in total. First, the figure shows that C# games have the highest percentage of cloned method and Java open source games have the lowest cloned method while C open source games fall in between. There are 1% of the methods in Java that are exact clones (i.e., type-1) while 5% of the methods are exact clones in C# and 2.5% are exact clones in C language. As the value of NiCad UPI increases, the cloned method percent increases for all programming languages. However, cloned method in C systems has a higher increasing rate than other Java and C#, and its cloned method percentage becomes larger than C# for UPI=0.3, which means it has more near-miss clones.

We also applied the statistical analysis of variance (ANOVA) test with $\alpha = 0.05$, to determine the cloning level between languages is significantly different. Therefore, we had chosen One Way ANOVA with the null and the alternative hypothesizes as follows.

H_0 : There is no statistical difference in cloning level between languages.

H_a : There is a significant difference in cloning level between languages.

ANOVA test with (F=1.74 < F_Crit=3.16), Accepts the null hypothesis, which means there is no statistically significant difference in cloning level between languages. Furthermore, we measured the ANOVA P-value between each pair of languages(Java, C#), (Java, C) and (C#, C) with $\alpha = 0.05$. The ANOVA p-values are 0.112, 0.027, and 0.829. We see that the p-values of (Java, C#) and (Java, C) are less than the alpha level selected ($\alpha = 0.05$). This means that there is no significant difference of TCMP between the languages Java and C# and the languages Java and C. But, the TCMP between the languages C and C# is statistically significant.

Table 3.3: NOCC, TCM, and TCLOC for all games.

		Freecol			robocode			Andengine			greenfoot			Terasology		
Java	UPI	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp
	0.3	141	372	2.65	160	621	9.1	98	284	5.92	64	149	3.9	149	492	6.96
	0.2	81	196	1.48	122	342	6.01	41	109	2.31	35	35	0.97	86	228	3.07
	0.1	37	80	0.67	64	167	3.44	20	46	0.99	17	35	0.97	57	127	1.66
	0	28	60	0.38	54	132	2.75	16	38	0.61	11	22	0.52	50	112	1.43
C#		Mono Game			Unity Steer			OpenRA			Samurai			Sleep Walker		
	UPI	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp
	0.3	210	553	8.6	2	4	2.74	26	62	1.44	7	18	3.89	66	165	7.1
	0.2	183	461	7.12	1	2	2.37	15	37	0.8	7	15	3.11	41	96	4.5
	0.1	146	372	5.76	1	2	2.37	8	22	0.49	5	11	2.37	26	62	2.87
	0	136	341	5.17	1	2	2.37	6	18	0.33	5	8	1.25	15	37	1
C		Freedriod			Berlios			Quake 3			Openarena			Egoboo		
	UPI	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp
	0.3	26	61	1.22	83	201	4.17	466	0.13	12.43	178	457	10.15	97	234	4.78
	0.2	4	9	0.15	44	93	2.12	360	0.1	9.96	130	294	7.68	53	114	3.23
	0.1	1	3	0.05	19	38	0.82	239	0.07	7.17	82	170	5.65	30	61	1.88
	0	1	2	0.03	8	16	0.17	147	0.04	2.11	55	110	2.5	17	34	0.51
		NetHack			Chocolate Doom											
	UPI	NOCC	TCM	TCLOCp	NOCC	TCM	TCLOCp									
	0.3	64	149	2.36	41	94	2.95									
	0.2	46	101	1.82	18	40	1.57									
	0.1	31	67	1.33	7	14	0.69									
	0	27	58	1.19	3	6	0.16									

ANOVA test indicates there is no statistically significant difference of cloning level between languages except C and C#. However, we still want to see if there are individual differences between games in the same language. In order to interpret these differences, we analyzed each individual system clones as shown in Figures 3.2, 3.3, and 3.4 (the TCMp values). An expected observation is that the existence of variation in cloning level for the same language games. But, this variation is still less than the variation between languages.

We also measured the ANOVA F-value for games in each individual language (Java, C#, and C games). We tested the following three null hypothesizes.

$H1_0$: No statistical difference in cloning level between Java games.

$H2_0$: No statistical difference in cloning level between C# games.

$H3_0$: No statistical difference in cloning level between C games.

The ANOVA F-value for Java games is 35.83 > F_Crit=3.06. The ANOVA F-value for C# is 17.27 > F_Crit=3.06. Also, the ANOVA F-value for C games is 2.78 > F_Crit=2.57. This means there is a statistically significant difference in cloning for games of the same language.

Figure 3.2 shows the TCMp for Java open source games. The results confirm the fact that the TCMp increases as UPI increases. Another interesting observation, there was a considerable variation of TCMp for Java games. As the results show, Robocode then Teraslolgy have a higher TCMp than other Java games.

Figure 3.3 shows the cloning level for C# open source games. As moving from identical clones (UPI=0) to near-miss clones (UPI>0), the number of total cloned methods increases. However, the rate of this increase is less than both Java and C open source games. The second interesting note is that Mono Game has a higher cloning level than all other games. We verified the clones of this game but nothing important. What is important is that MonoGame is the largest game in the C# set. On the other hand, OpenRa is the second largest game in the set, and it has the lowest cloning level, which refutes the fact that there is a relationship between the game size and cloning level.

We observed two facts from Figure 3.4. First, the clone level between C games in more varied from that Java and C# games. Second, not all C games have the same rate of near-miss cloning level. The games Quake 3, OpenArena, and Egoboo have a higher level of near-miss clones, which participate in increasing the C games cloning level that appeared in Figure 3.1. Besides, we observed that NetHack has a lower level of near-miss clones than all C open source games.

Figures 3.5, 3.6, and 3.7 show the TCLOCp for each system in the target programming languages. In these figures, the general trend is the same as in the corresponding TCMp figures. These figures emphasize the same facts that have been concluded from TCMp metric. There is a variation in cloning levels across programming language games. In addition, there is a variation in cloning level for the same language games. But, this variation is still less than the variation between languages. Both TCMp and TCLOCp have the same order of games according to their cloning level. Figure 3.8 shows the TCLOCp by language, and Table

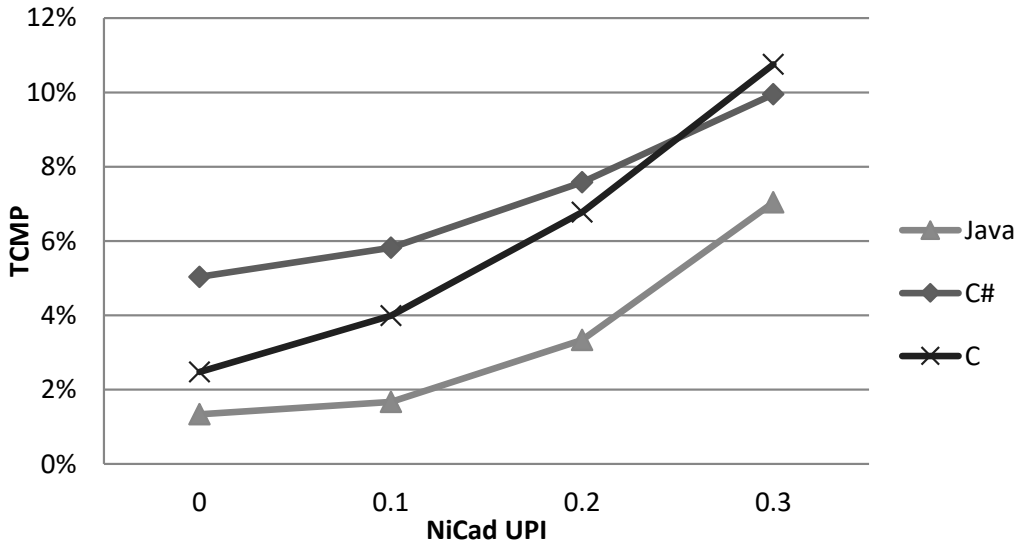


Figure 3.1: TCMp by language.

3.8 shows the detailed value of TCM, TCLOCp, and NOCC for each game.

Quake and OpenArena have a relatively high cloning level in C games. Besides, these two games have higher, similar trend, near-miss clones than other C games. The interpretation behind the identical trends of these games is that they belong to the same engine series. Therefore, we are analyzing the game engine series in a separate section to complete our observations.

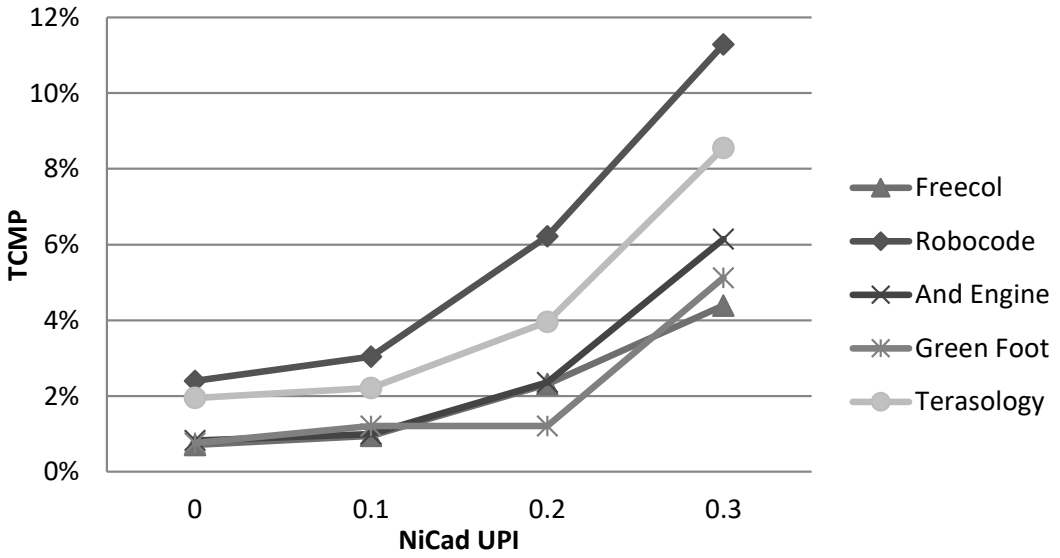


Figure 3.2: TCMp for Java games

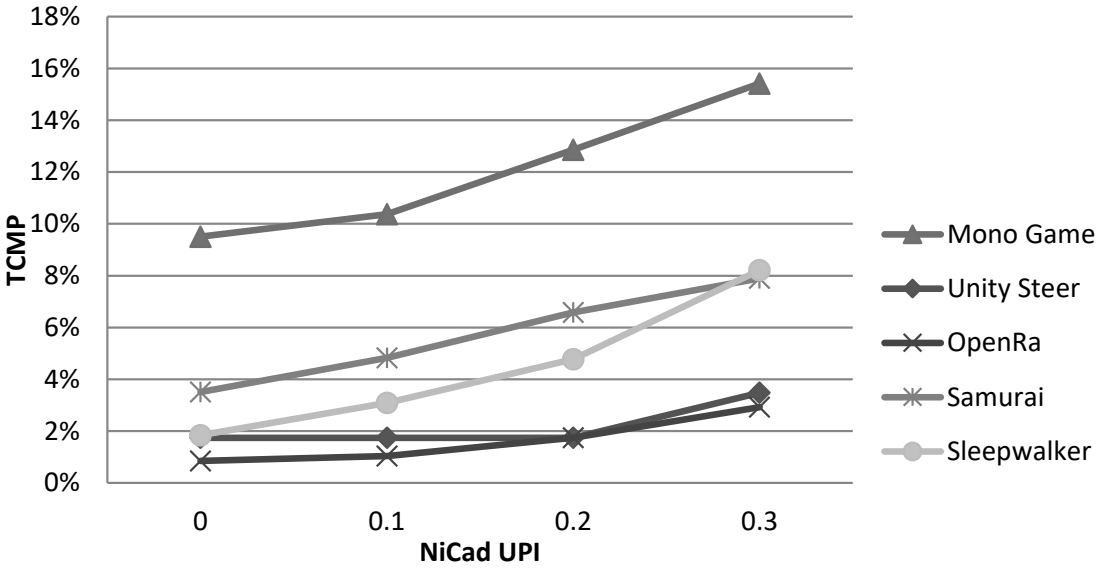


Figure 3.3: TCMp for C# games

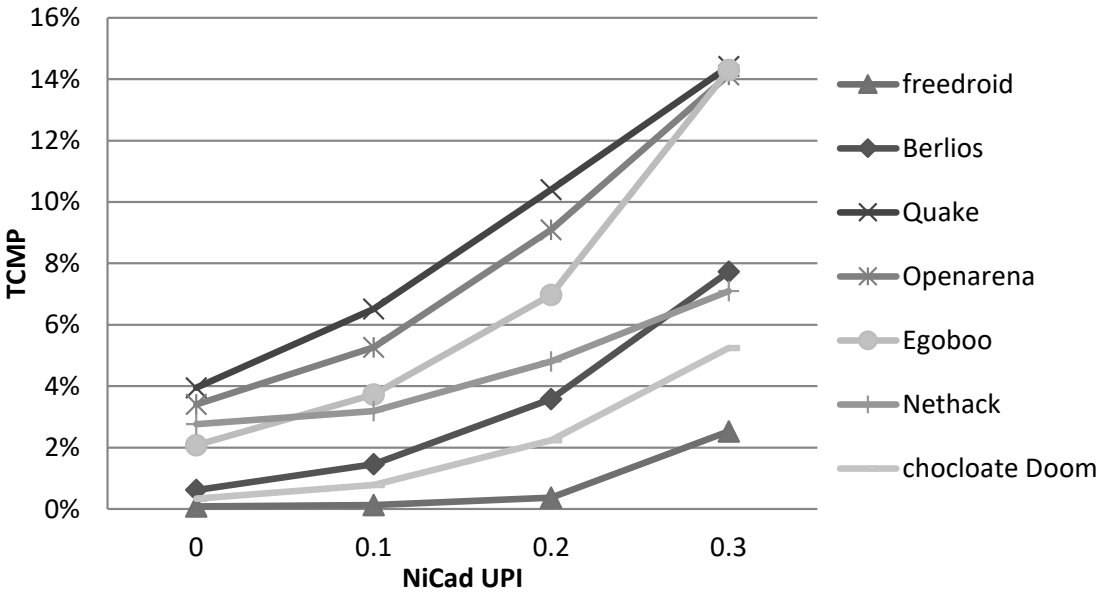


Figure 3.4: TCMp for C games

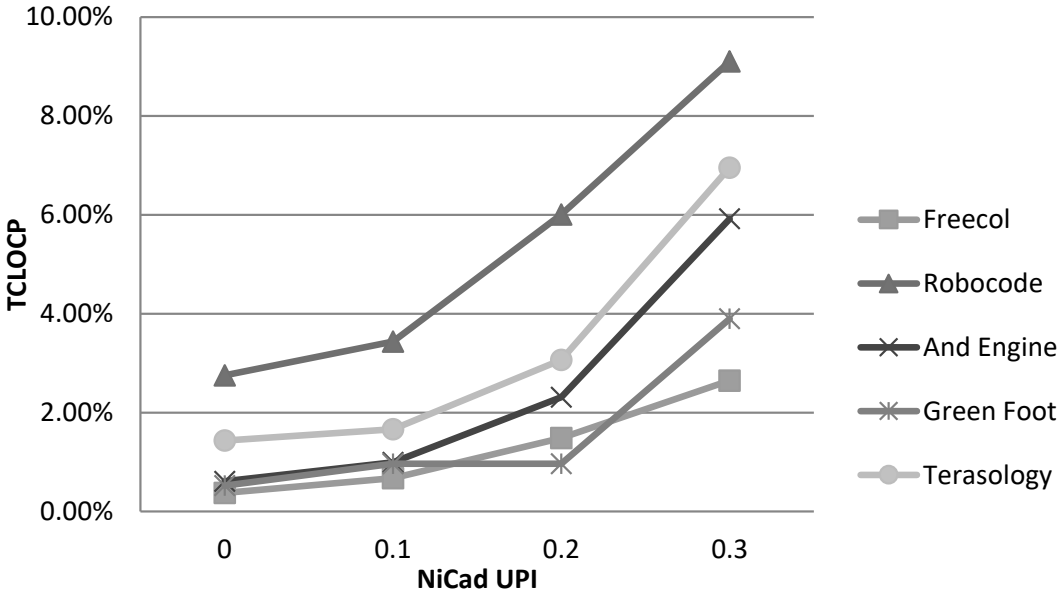


Figure 3.5: TCLOCp for Java games

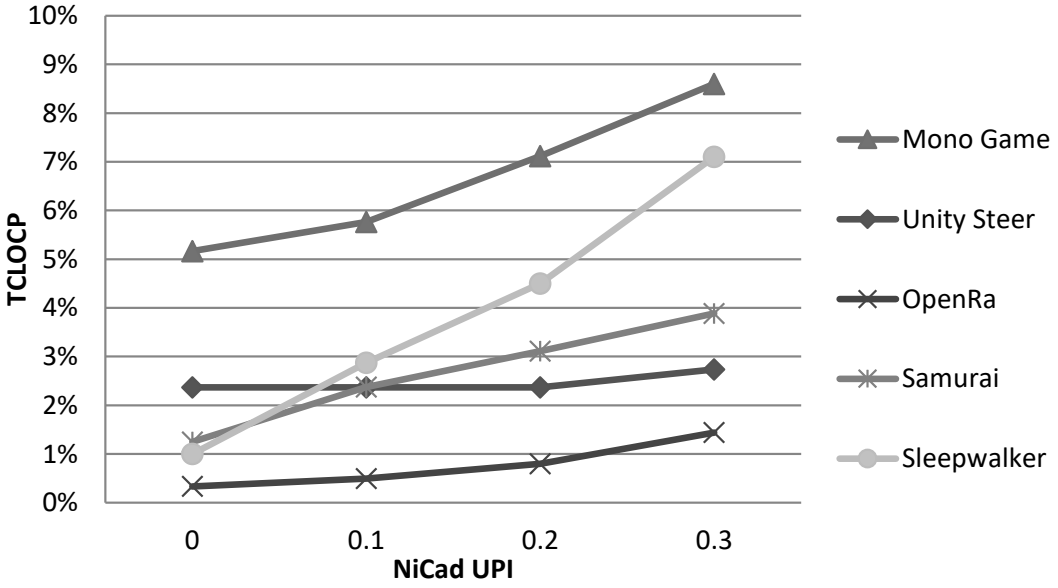


Figure 3.6: TCLOCp for C# games

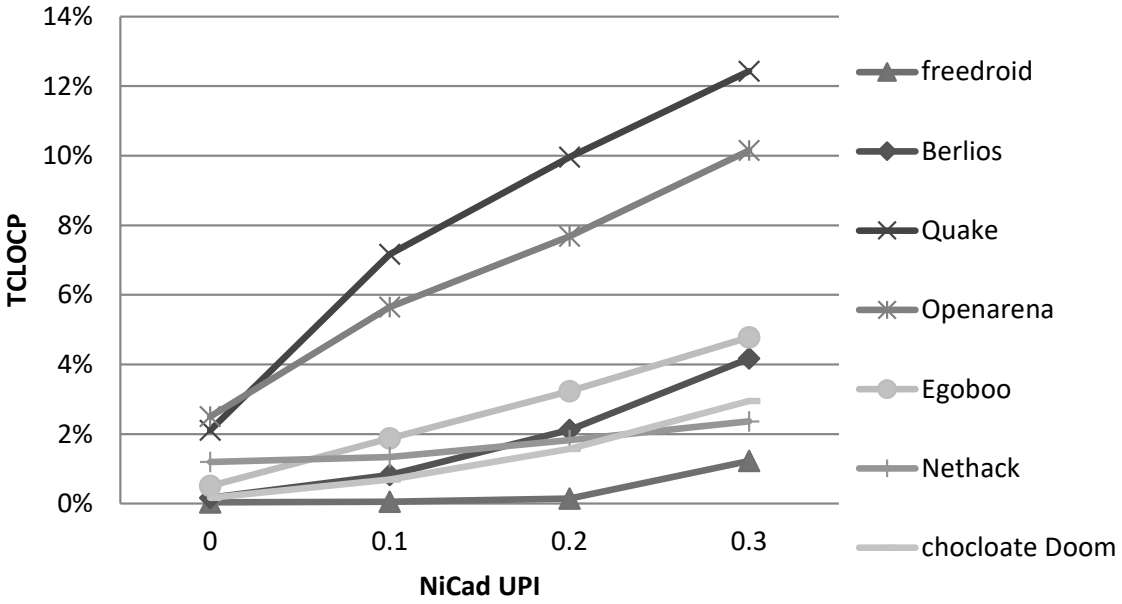


Figure 3.7: TCLOCp for C games

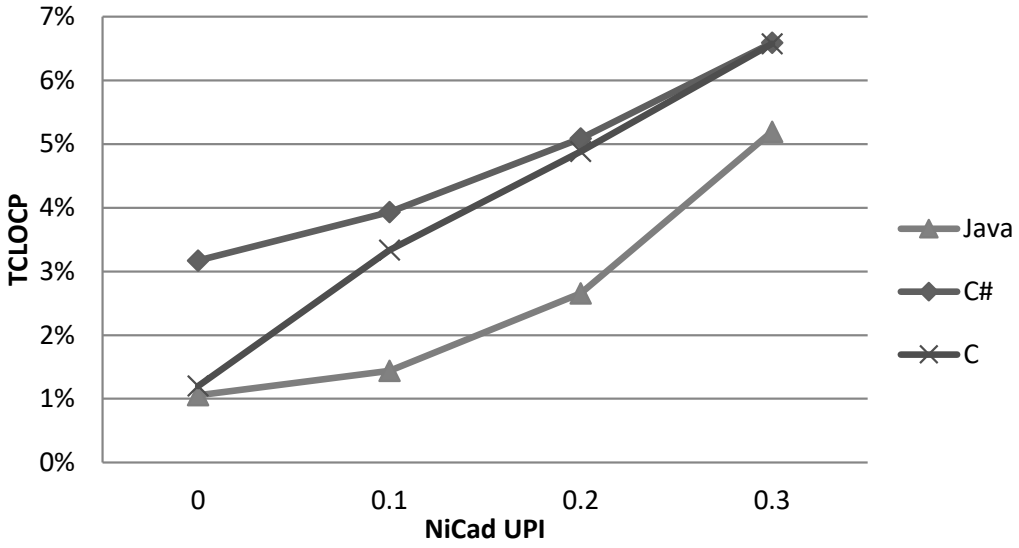


Figure 3.8: TCLOCp by language

3.5.1.2 Clone Associated Files

The second metrics we used to analyze clones is FAWC. As a quality measure, the smaller the FAWCp, the better system quality is. Hence the less FAWCp, the less scattered clones in the system is and consecutively less maintaining effort. Figure 3.9 shows the average FAWCp by language, and Table 3.4 shows the detail values of FAWC and FAWCp for each game. Identical clones are more spread in C# games followed by C games and least spread in Java games. By looking at TCMp in Figure 3.1, we have the same language order according to cloning level. The interesting thing about comparing these figures (Figure 3.1 and Figure 3.9), is that the relationship between TCMp and FAWCp does not follow as we move toward near-miss clones. Near-miss Java clones spread out more rapidly in game files and exceed the FAWCp in C#. In other words, the near-miss clones in C# games reside in the same files that contain identical clones. Even C# games have the highest percent of FAWC for identical clones that indicate the game quality.

By looking at FAWCp for individual games, we find that the relationships between TCMp and FAWCp do not hold. For example, in Java games, RoboCode indeed has the highest TCMp and FAWCp. On the contrary, FreeCol has the lowest TCMp and the second-highest FAWCp and TeraSology, the second-highest TCMp, has the lowest FAWCp. Also, this fact emphasizes by the other two languages, C# and C.

The other fact we concluded from Table 3.4 is that FAWCp is more condensed among the same language games more than across language games. However, there is still a reasonable variation in the same language games.

Table 3.4: FAWC and FAWCp by language.

Language			FAWC				FAWCp			
	Game	NoF	0.3	0.2	0.1	0	0.3	0.2	0.1	0
Java	Freecol	689	170	110	53	44	24.67344	15.96517	7.692308	6.386067
	Robocode	653	201	159	96	77	30.78101	24.34916	14.70138	11.79173
	AndEngine	596	102	49	28	21	17.11409	8.221477	4.697987	3.52349
	Greenfoot	345	64	39	17	12	18.55072	11.30435	4.927536	3.478261
	Terasology	614	93	46	22	15	26.95652	7.491857	3.583062	2.442997
	Total	2897	630	403	216	169	21.74663	13.91094	7.455989	5.833621
C#	Mono Game	791	228	193	158	145	28.82427	24.39949	19.97472	18.33123
	Unity Steer	51	4	2	2	2	7.843137	3.921569	3.921569	3.921569
	OpenRA	630	49	30	18	16	7.777778	4.761905	2.857143	2.539683
	Samurai	82	7	7	5	5	8.536585	8.536585	6.097561	6.097561
	Sleep Walker	528	84	58	42	27	15.90909	10.98485	7.954545	5.113636
	Total	2082	372	290	225	195	17.86744	13.92891	10.80692	9.365994
C	Freedroid	128	25	6	3	2	19.53125	4.6875	2.34375	1.5625
	Berlios	188	54	35	21	10	28.7234	18.61702	11.17021	5.319149
	Quake 3	428	209	157	102	61	48.83178	36.68224	23.83178	14.25234
	Open Arena	171	75	62	35	17	43.85965	36.25731	20.46784	9.94152
	Egoboo	179	37	24	8	2	20.67039	13.40782	4.469274	1.117318
	Nethack	254	42	32	16	11	16.53543	12.59843	6.299213	4.330709
	Chocolate Doom	174	38	20	7	4	21.83908	11.49425	4.022989	2.298851
	Total	1522	480	336	192	107	31.53745	22.07622	12.61498	7.030223

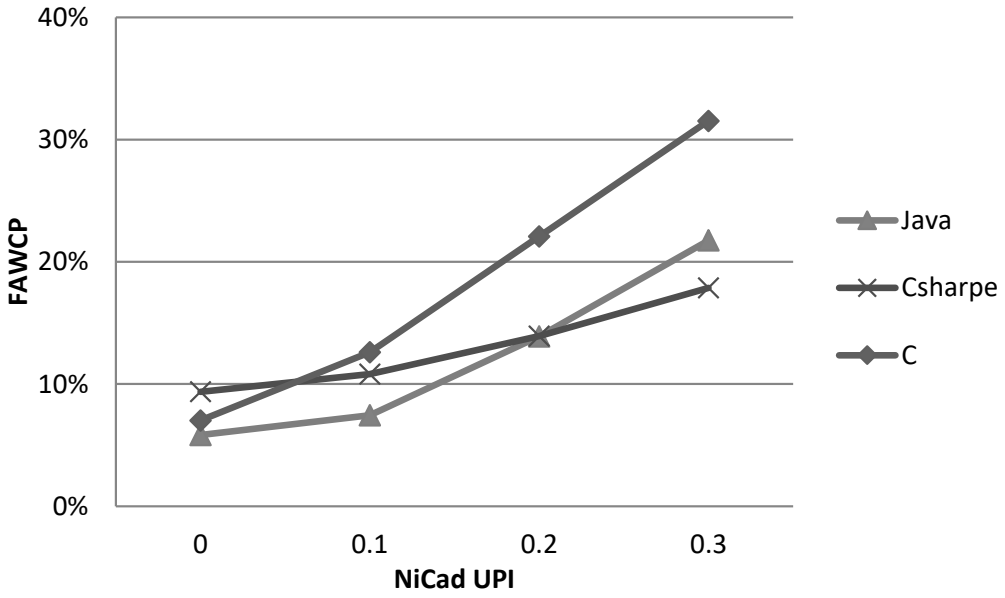


Figure 3.9: FAWCp by language

3.5.1.3 Profile of Cloning Localization

As mentioned earlier, the better the system quality, the less cloning level is, and fewer files contain clones. Furthermore, in this section, we discuss a metric for clone localization. As a measure of quality, the more localized the clones are, the better quality of the system is, leading to less maintenance efforts and refactoring. We are using Clone Class Radius (CCR) as a measure of cloning localization. CCR is calculated for each clone class in the system as described in Section 3.4. The average for all clone class is considered the average CCR for the system.

Figure 3.10 compares CCR by programming language through different UPI thresholds. Basically, an important factor that affects the value of the CCR is the nature and the size of the system's hierarchical structure. In our experiment, we had chosen different size systems for each programming language to avoid the potential bias due to system size. The CCR value for small systems that have only one folder would be equal to zero. Two facts are to discuss in Figure 3.10. First, the variation between CCR for programming languages. C games have smaller CCR. One important factor affecting is that C games are smaller than other Java and C# games.

The second and interesting fact is the average class's radius for identical clones is larger than near-miss clones, which means near-miss clones reside on the same locations as identical clones. We also analyzed the CCR for each game individually, and we find a high variation of CCR for games of the same programming language. The main facts to conclude from this subsection are: (1) clones in C games are more localized than clones in C# and Java games, and (2) CCS has an inverse relationship with CCR.

Summary of answer to RQ1: *There is differences in cloning density among the open source games*

with respect to programming languages. Our result reveals that games developed in C have more clones than games developed in C# and Java. However, this variation is not statically significant. Also, there is a clear difference in cloning density among games of the same programming languages. In addition, games developed in C have more percentage of files associated with clones and those files are more localized as compared to the games in other languages we analyzed. This suggests that game developers should be aware of the clones, and their impacts on the systems in general, and the developers using C language for games may need additional cautions.

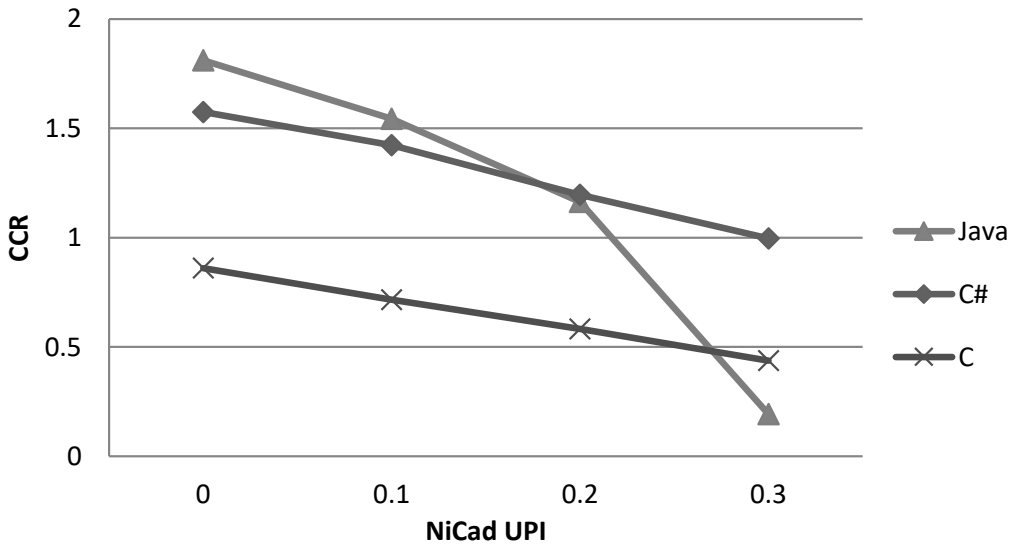


Figure 3.10: Average CCR by language.

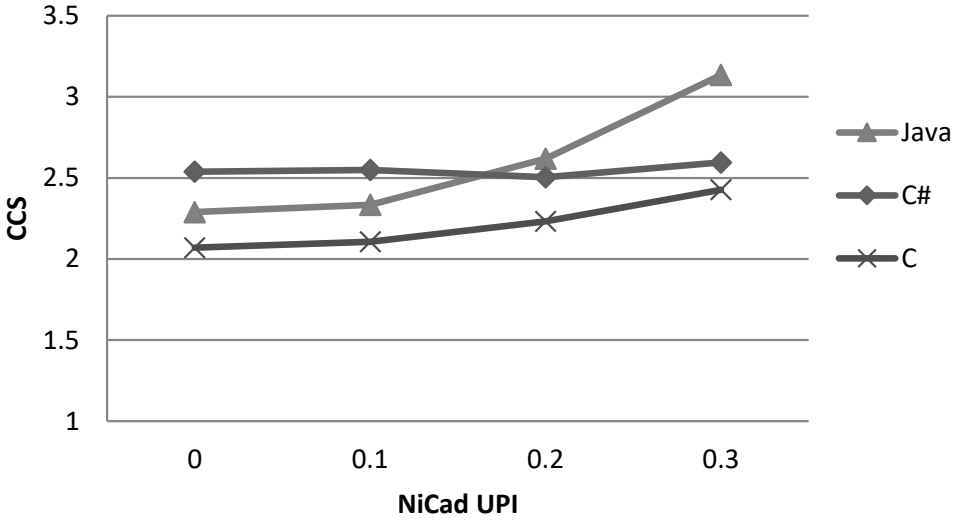


Figure 3.11: Average CCS by language

3.5.2 Game vs. Non-Game

To answer ***RQ2***, we compared cloning status in games with that of non-game systems. We chose three popular open-source, non-games for each programming language. Originally, these systems were used by Roy and Cordy's [136] empirical study on clones. Table 3.2 shows these systems. We applied the same metrics as the previous section. Table 3.5 shows the metrics for these systems and the corresponding metrics for open source games. The most interesting observation is that, in general, the cloning level for all open source non-games is higher than all open source games. Also, it shows the CCS is larger in non-game open source, and the CCR is smaller that shows games have better quality in terms of cloning level and distribution in thve file system.

Starting from C language, we chose three open source non-games (Appache, Postgresql, and Cook) and all C games used in previous section. The three used cloning level metrics, TCMp, TCLOCp, and FAWCp, indicate that there are more clones in open source non-games. The total TCMp for non-games open source is 13.31%, while the corresponding TCPM for games is 10.75%. The TCLOCp for non-game open source is 7.64%, while the corresponding value for games is 6.65%. Also, the FAWCp for non-games is 34.12%; the corresponding value for games is 31.54%. On the other hand, clones in non-games are more localized than in games open source. The average CCR for all non-games open source is 0.37, while it is 0.44 in games. The relationship between CCR and CCS is holding in this section, too, as the average CCS for non-games is higher than the CCS for games, and the CCR is smaller for non-games.

In Java, we chose three non-games open source (Eclipse JDT core, CIRC and JHotdrow), and we used the Java games used in the previous section. The results of comparing cloning status between game and

non-games open source have a similar trend as C language. Table 3.5 shows that the cloning level in non-game open source is higher than in game open source. On contrary to C language, the Java clones are more localized than in non-games even the CCS is larger in non-game open source.

We use mono C# compiler, Castle, and Nant as C# non-game open source. Even the results in Table 3.5 show that there are more clones in game open source. But, these clones come from one system, Monogame. If we ignore this system, then the values of TCMp, TCLOCp, and FAWCp would be 5.57%, 4.14%, and 11.15%, respectively, which mean there are more clones in non-games. The value of CCR of non-games is 0.42 and is smaller than the game's CCR. Besides, the value of CCS for non-game is larger than game's CCS.

Summary of answer to RQ2: *First, results for the three cloning level metrics used (TCMp, TCLOCp and FAWCp) indicate that there are more clones in open source non-games systems as compared to open source games. Second, clones in non-games are more localized than in games open source. The average CCR for all non-game open source systems is 0.37 while it is 0.44 in games. Finally, there is no clear evidence that games have less files associate with clones.*

Table 3.5: Games vs. non-games clone status.

		System	NOCC	NOM	TCM	TCMp	LOC	TCLOC	TCLOCp	NOF	FAWC	FAWCp	AvgCCR	AvgCCS
C	Non-Games	Apache http	138	3878	341	8.79	200K	11666	5.82	252	73	28.97	0.54	2.47
		Postgresql	3255	15399	2204	14.31	1018K	81939	8.04	1155	398	34.46	0.37	2.91
		Cook	74	1335	198	14.83	73K	5229	7.07	296	110	37.16	0.18	2.68
		All	3467	20612	2743	13.31	1293K	98834	7.64	1703	581	34.12	0.37	2.89
	Games	Freedroid	26	2414	61	2.53	77K	951	1.22	128	25	19.53	0.04	2.35
		Berlios	83	2601	201	7.73	162K	6790	4.17	188	54	28.72	0.24	2.42
		Quake	466	7798	1124	14.41	348K	43271	12.43	428	209	48.83	0.68	2.41
		Openarena	178	3232	457	14.14	147K	15014	10.15	171	75	43.86	0.32	2.55
		egoboo	97	1636	234	14.3	196K	9413	4.78	179	37	20.67	0	2.41
		Nethack	64	2100	149	7.1	222K	5248	2.36	254	42	16.54	0.23	2.33
		Choclate Doom	41	1792	94	5.25	103K	3062	2.95	174	38	21.84	0.22	2.29
		All	955	21573	2320	10.75	1259K	83749	6.65	1522	480	31.54	0.44	2.43
Java	Non-Games	Eclipse jdt Core	709	18257	2283	12.5	455k	49773	10.94	1202	479	39.85	1.1	3.22
		CIRC	13	765	33	4.31	12k	361	2.82	65	15	23.08	0.62	2.54
		JHotDraw	56	2886	173	5.99	189k	1980	1.05	469	70	14.93	0.45	3.09
		All	778	21908	2489	11.36	657k	52114	7.93	1736	564	32.49	1.04	3.2
	Games	Freecol	141	8472	372	4.39	193k	5113	2.65	689	170	24.67	0.18	2.64
		Robocode	160	5502	621	11.29	94k	8583	9.1	653	201	30.78	2.68	3.88
		AndEngine	98	4621	284	6.15	66k	3926	5.92	596	102	17.11	0.26	2.9
		GreenFoot	64	2907	149	5.13	60k	2346	3.9	345	64	18.55	0.2	2.33
		Terasology	149	5754	492	8.55	86k	6034	6.96	614	93	18.55	0.2	2.33
		All	612	27256	1918	7.04	500k	26002	5.19	2897	630	21.75	0.85	2.9
C#	Non-Games	mcs	20	1993	52	2.61	88k	1065	1.21	54	14	25.93	0	2.6
		Castle	329	9236	908	9.83	270k	18496	6.84	2407	302	12.55	0.51	2.76
		nant	84	2335	216	9.25	104k	4380	4.18	429	88	20.51	0.15	2.57
		All	433	13564	1176	8.67	463k	23941	5.17	2890	404	13.98	0.42	2.72
	Games	monogame	210	3587	553	15.42	166k	14301	8.6	791	228	28.82	1.33	2.63
		Unity Steer	2	115	4	3.48	5k	141	2.74	51	4	7.84	0	2
		OpenRA	26	2121	62	2.92	64k	926	1.44	630	49	7.78	0.73	2.38
		Samurai	5	228	18	7.89	5k	221	3.89	82	7	8.54	0	3.6
		Sleepwalker	66	2010	165	8.21	61k	4379	7.1	528	84	15.91	0.14	2.5
		All	309	8061	802	9.95	303k	19968	6.59	2082	372	17.87	1	2.6

3.5.3 Clones across Game Engines

In this section we answer ***RQ3*** by analyzing cloning status across game engines to show the amount of shared code in game open source and whether game developers are used to copy code from another game. The experiment in this section is divided into three parts according to engine selection. We studied clone status between unrelated engines, engines of the same game genre, and recreated games.

3.5.3.1 Clones across Unrelated Game Engines

The purpose of this experiment is to study the clone status across unrelated -not belonging to the same category games. Therefore, we chose two unrelated game engines for each programming language. For Java, we had chosen Env3D and Jake2 Game engines. We selected FreeCiv2.3.3 and SGE2D for our C language case study. Finally, we chose NetGore and Axiom game engines for C#. Our experiment started with an analysis of clones for each game independently. Then we combine every two games of the same programming language as one system and detect and analyze its clones. Thus, the number of cross game clones can be defined as clones for combined games A and B -(clones of game A + clones of game B).

Table 3.6 summarizes the results for this experiment. The results indicate there are no cross game engine clones for all cases that we had studied. For C game engines there are 1184 clone pairs where 1175 from the first engine, 9 from the second engine and no clone pair exist between the two engines. A total of 737 clone pairs in Java game engines 20 pair come from the first engine, and 717 come from the second engine. In the same manner C# game engines contain 1734 clone pairs that came either from the first engine or the second engine and no cross game engine clone pair.

Table 3.6: clone status across unrelated games

	Game	NOCC	NOCP	TCM	TCMp
C	FreeCiv	342	1175	905	7.99
	SGE	9	9	18	5.26
	Both	351	1184	928	7.95
Java	Env3D	16	20	34	4.59
	Jake2	246	717	660	15.87
	Both	259	737	694	14.16
C#	NetGore	91	326	235	11.66
	Axiom Engine	163	1408	506	11.84
	Both	254	1734	741	11.78

3.5.3.2 Same Genre Game Engines Clones

In the previous section, we studied clone status between unrelated game engines. Thus, the next question to answer is: what is the clones' status between game engines that belong to the same category. To answer this question, we have to find a number of open source game engines that belong to the same category. Id Software [2] developed a number of game series that all classified first-person shooter. I started from Vintage series through Wolfenstein, Doom, Quake, and Enemy Territory series and ended by mobile series. Each of the mentioned series has a number of games. Thus, we select one game from each series for this experiment. We select Hexan from Vintage series, Wolf3s from Wolfenstein series, Chocolate Doom from Doom series, Quake 2 from Quake series, and Enemy Territory from Enemy territory series.

We did a similar experiment to the previous section. We first analyzed clones of each game independently. Then we combine every two games that belong to a consecutive series as a single dataset and detect and analyze its clones. Table 3.7 shows the clone status in each individual game and the results of combined games.

Tracing Table 3.7 not only shows how cloning Software games engine occurs across Id but also it shows the program evolution in Id Software games. It shows that the next engine usually larger in terms of LOC and number of methods except for Enemy Territory engine. Furthermore, the cloning level is more in the successor engines in terms of TCMp and TCLOCp except for Enemy Territory game. Also, CCR increased in successor game, and its relationship with CCS is preserved as well. We are interested in number of cross-engine clones. An interesting finding is that the first three engine pairs show there are no cross-engine clones in spite they are of the same category (first-person shoot) and of the same vendor. However, there are a limited number of clone pairs, 117 clone pairs, which is 5.5% of the total number of clone pairs, shared between Quake2 and Enemy Territory games. This implies there is no or limited clones exist between games of the same family. This is an interesting and valuable observation since they are holding a low cloning level across systems comparing to cloning level in non-game open source [8, 9, 21, 28, 29].

Table 3.7: Clone status for same category game engines

Game engine	NOCC	NOCP	NOCP cross games	TCM	TCMp	TCL OCp
Hexan	35	78	0	95	7.77	3.87
Wolf3D	22	36		50	8.97	9.37
Both	57	114		145	8.1369	4.84
Wolf3D	22	36		50	8.97	9.37
Chocolate Doom	40	60		89	5.03	3.44
Both	62	96	0	139	5.9682	4.33
Chocolate Doom	40	60		89	5.03	3.44
Quake 2	601	1027		1337	29.89	32.05
Both	641	1087	0	1426	22.841	21.59
Quake 2	601	1027		1337	29.89	32.05
Enemy Territory	465	1000		1116	12.27	8.49
Both	1072	2144	117	2511	18.5067	14.36

3.5.3.3 Clones across Recreated Games

In previous sections, we have shown the clone status across games that are not related and games that belong to the same genre. Open Source community and game market place are full of cloned or recreated games. Do these games share a common code clones base? In this section we are studying clone status in recreated games. In general, Id Software games are large series systems. For example, Quake series have 15 main versions in the main series starting from the main quake game that appears in 1996 and ending by quake Xbox game. However, quake family contains large number of games that are derived from the main series. In this section we study clone status in Quake series from id Software. In this section we are discovering two facts 1) do game series members have a similar clone status. 2) What is the clone status that is shared between games series members?

We selected four games from quake games as our first dataset for this study. Table 3.8 shows the clone status in four Quake Games where QW and WinQuake are two Quake I Games and Quake 3 game is called Quake 3 Arena. As the table shows that these four games have a similar clone status, TCMp, TCLOCp, FAWCp and CCS. However, the table indicates that Quake II has the highest clone percent and this percent reduced in Quake III. This shows that game series have kind of similar clone status as of Point 1 above.

As regards to Point 2 above, we detect clones in these games pairwise. We selected three game pairs as shown in Table 3.8. The values of TCMp and TCLOCp are much higher than the values in single game. For example, TCMP for QW is 21.01% and for WinQuake is 19.16%. But for two games together, it is 52.59%. It is clear that TCMp and TCLOCp increment due to cross games clones. The fourth column in the table shows the number of cross-games clone pairs. These numbers represent a high cloning level if we are

comparing it to the number of clone pairs in each individual game. Also, we conclude that there are more clones in consecutive games in the series based on Table 3.8. For example, the highest cloning level exists in QW game and WinQuake which is Quake 1. Another example, the combination of Quake1 and Quake 2 games has more clones than the combination of Quake 1 and Quake 2 games.

Our study shows that there are lots of clones between series games. It seems reasonable because the games in the series are evolved (cloned) from each other. In such cases where there are common functionalities across game series or recreated games, it would be beneficial to encapsulate those functionalities into a library. This library would be useful for other game series or other game developers.

Table 3.8: Clone status across recreated games

System	NOCC	NOCP	NOCP cross games	TCM	TCMP	TCL OCp
QW	148	257		374	21.01	15.01
WinQuake	177	268		399	19.16	14.14
Both	837	1792	1267	2031	52.59	47.86
WinQuake	177	268		399	19.16	14.14
Quake 2	596	1039		1328	29.66	32.27
Both	879	1630	323	2028	30.92	47.86
Quake 2	596	1039		1328	29.66	32.27
Quake 3	466	1018		1124	14.41	12.43
Both	1079	2288	229	2028	30.92	19.11

Summary of answer to RQ3: *There is a clear evidence that there is less practice of copying source code across games open source. However, the cloning density is very high between games in the same series.*

3.5.4 Code Normalization and Code Clone Detection

In our previous experiments, we used NiCad for the purpose of detecting clones. NiCad has been proven to detect Type-1, Type-2 and Type-3 clones with high precision and recall for the selected UPI thresholds [160, 155, 156, 160, 157]. Besides, NiCad is empowered with many code transformation options that enable itto detect more and different types of clones. However, we did not apply any of these code transformations in the previous experiments as we focus on clones resulted by copy and paste. Also, code normalization could affect the accuracy of the detection results. The goal of this study is not to evaluate NiCad rather to explore type of clones NiCad can detect after applying certain code transformations.

In this section, we selected three games, FreeCol for java, FreeDroid for C and MonoGame for C#. We Run NiCad at 4 different thresholds 0.0, 0.1, 0.2 and 0.3. First, we used NiCad without any transformation (Base). Then we used one transformation at a time. Tables 3.9, 3.10 and 3.11 show the number of clones

detected in each system before any code transformation and after code transformation. Also, these numbers are drawn in Figures 3.12, 3.13 and 3.14.

Table 3.9: Number of clone pairs detected in Java system using different transformation

UPI	0	0.1	0.2	0.3
Base	37	48	169	377
Filter Declaration	30	41	125	315
Abstraction Block	23	24	152	280
Abstraction Declaration	30	41	116	303
Abstraction literal	47	58	207	465
Abstraction condition	40	58	207	680
Abstraction Expression	360	532	10322	27530
Consistent Rename	105	122	179	624
Blind Rename	123	156	518	1285

Table 3.10: Number of clone pairs detected in C system using different transformation

UPI	0	0.1	0.2	0.3
Base	1	3	6	48
Filter Declaration	1	2	27	49
Abstraction Block	0	0	4	38
Abstraction Declaration	1	1	13	19
Abstraction literal	1	3	8	57
Abstraction condition	1	7	15	71
Abstraction Expression	3	11	34	255
Consistent Rename	28	30	49	82
Blind Rename	28	36	94	151

The results show that the code transformations (filtering declarations, abstraction of blocks, and abstraction of declarations) reduce the number of detected clones. It means that declarations and blocks hold a good portion of similar contents between clones and eliminate them results diverge some clones. On the other hand, abstraction of literals, abstraction of conditions, abstraction of expressions, consistent renaming and blind renaming enhance clone detection.

To have a better understanding of the types of clones that are resulted by using code transformation, we collected all the extra clones that are resulted by using each type of transformation. Then we manually validate a random 50 clone pairs for each kind of transformation. We have found that each type of code transformation enables NiCad to detect extra Type-2 or Type-3 clones. For example, abstraction of conditions

Table 3.11: Number of clone pairs detected in C# system using different transformation

UPI	0	0.1	0.2	0.3
Base	411	478	579	791
Filter Declaration	372	438	573	734
Abstraction Block	193	225	301	474
Abstraction Declaration	372	438	567	709
Abstraction literal	440	520	633	833
Abstraction condition	429	482	599	881
Abstraction Expression	478	589	1012	1452
Consistent Rename	512	562	645	803
Blind Rename	523	616	798	1115

enables detecting extra Type-3 clones that contain conditions, if and switch statements. Also, we notice that the abstraction of expression enables the detection of a larger number of extra clones. The manual verification of these clones shows that the majority are larger methods. Finally, the manual validation shows that abstraction of expression produces more false-positive clones.

Summary of answer to RQ4: *Using the code transformations, filter declaration, abstract declaration, and abstract block results in missing some clones. But, using the code transformations, abstract literal, abstract condition, abstract expression, consistent rename, and blind rename enables the detection of more Type-2 and Type-3 clones. In general, code transformation enables the detection of more code clones and produces false-positive clones.*

To answer ***RQ5***, we conduct a similar experiment in RQ4; we run NiCad with multiple code transformation then we manually investigate types of extra clones detected. In this experiment, we consider only code transformations that produced extra clones only. Also, Not all code transformations could be combined together at the same time. For example, both consistent rename and blind rename could not apply together, and filter declaration and abstract declaration could not apply together.

Table 3.12 shows the code transformation combinations used and the number of clone pairs detected in the Java game (FreeCol). To evaluate what type of clones detected by combining two code transformations, we extracted the extra clones detected by combining two code transformations. Let *Set(t)* is the set of clones detected by applying the code transformation *t*. Then the extra clones detected by applying the combined transformations *t1* and *t1* are:

$$Extra - Set(t1, t2) = Set(t1, t2) \nsubseteq (Set(t1) \cap Set(t2)) \quad (1)$$

We randomly selected 100 extra clone pairs that were detected by combining two code transformations. We classified them into Undecided, Type-1, Type-2, Type-3, Type-4 and false positive. We labelled clones

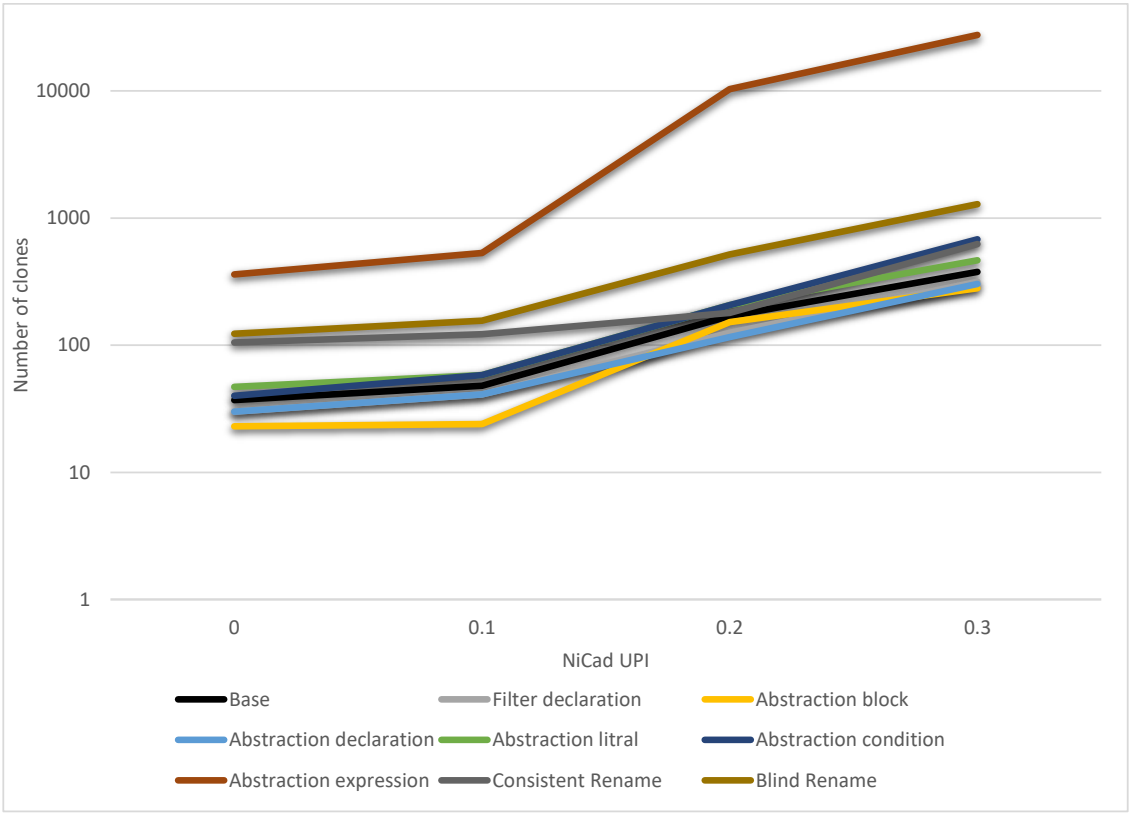


Figure 3.12: Number of clones detected in Freecol using different transformation

as Type-1, Type-2 or Type-3 according to their syntactic similarity. Semantically similar clones are labelled as Type-4. Some clones hold a similar structure but not syntactic similar enough to be classified as Type-1, Type-2 or Type-3; at the same time, they do not perform the same functionality. These clones are classified as undecided. Manual validation indicates that 13% are classified as undecided, 61% are classified as Type-3, 7% are classified Type-4, and 19% are false positive. Also, we notice that abstraction of expression produced the most false positive clones.

Summary of answer to RQ5: *Applying more than one type of code transformation to the source code helps in detecting more Type-3 and Type-4 clones. On the other hand, it produces more false clones.*

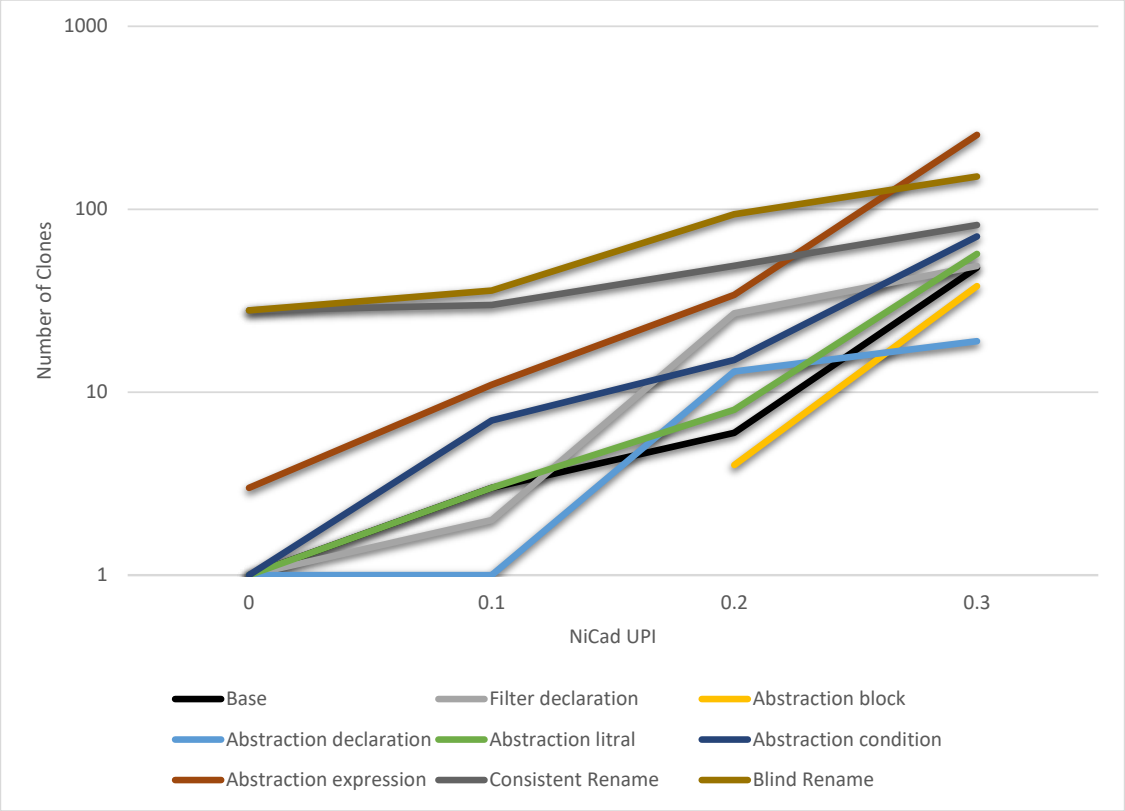


Figure 3.13: Number of clones detected in Freedroid using different transformation

Table 3.12: Number of clone pairs detected in Java system using combined transformation

UPI	0	0.1	0.2	0.3
Base	37	48	169	377
Consistent Rename & Abstraction Literal	105	122	290	624
Consistent Rename & Abstraction Condition	120	138	383	1302
Consistent Renamee & Abstraction Expression	1189	1387	12565	35363
Blind Rename & Abstraction Literal	306	352	859	1931
Blind Rename & Abstraction Condition	153	191	750	2369
Blind Rename & Abstraction Expression	1373	1727	16351	50036

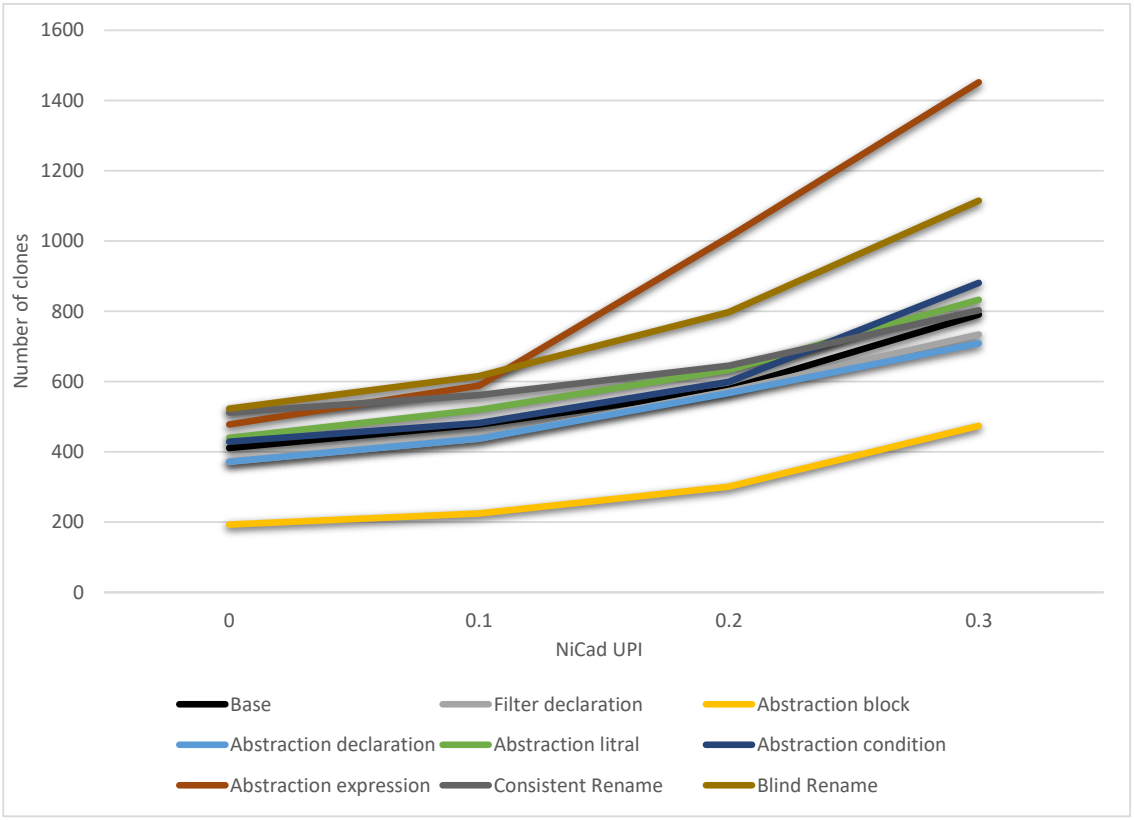


Figure 3.14: Number of clones detected in Monogame using different transformation

3.6 Related Work

Empirical studies of clones in open source has been a popular topic, and there have been a great many studies [137]. For example, when a new tool is published, it comes with an empirical study for validating the corresponding tool [141, 137]. Empirical studies of clones have also been conducted as part of tool comparison experiments such as the Bellon's benchmark [24]. Kapser and Godfrey have conducted several studies on clones, e.g., for deriving a taxonomy of clones [84] and then studying the harmfulness / usefulness of clones [86]. There are also a number of studies that investigate cloning and their evolution for deriving different maintenance implications of clones [137].

AL-Ekram et al. [10] conducted an empirical study of source code cloning across software systems. They analyzed clones between software systems in two domains: Text Editors and Window Managers. They used CCFinder as clone detector for Type-1 and Type-2 clones. Their results indicate that, there are not as many clones across systems and most of these clones are accidental rather than copy and paste clones. On the other hand, Ishihara et al. [72] did an empirical study for inter-project method clones for Java projects. Their method is based on hashing the generated value from AST (Abstract Syntax Trees). Although they did not consider domain-related analysis of the methods and they investigated only the top 100 shared method sets, their judgment indicates a 56 cloned method set is suitable as a library candidate.

There have also been a few studies that partly used open source games. For example, Gabel and Su [54] used four games out of 30 systems to study uniqueness of source code. Another clone related study that uses games is presented by Schulze and Apel [145]. They conducted an empirical study to analyze the relationship between feature-oriented programming and clones. Furthermore, some studies [111, 54, 87, 112, 22, 68] used the source code of Freecol game engine as part of their target systems.

Munro et al. [126] analyzed the structure of Quake game engine series. Similar to our work, part of their study was detecting how much shared code was between Quake 1 and other Quake games. For this purpose, they use Simian to detect clones between Quake1 and other Quake games. They found that the amount of shared code decreases between Quake 1 and consecutive Quake game engines. But, the purpose of their study is to analyze the structure and evolution of Quake games.

Ciancarini and Favini [32] detect chess games that are cloned or derived from other chess games. Their approach was based on performing tournaments between tested games and candidate games are gone through code clone analysis, then the final decision is made by expert referees. Their work shows the importance of code clone analysis in detection of cloned or plagiarized games.

Chen et al. [28] studied a number of open source games and is related to ours to some extent. However, again they focused on studying different types of clones such as the presence of Type-1, Type-2 and Type-3 clones in games. Our focus is on the other hand, to study the intentional copy/paste and reuse of program code between games and game engines. For detecting, Type-2 and Type-3 clones, one needs to normalize the source code before comparison, and thus the results may not reflect the reuse of code fragments among the

subject software systems. As noted earlier, we use pretty-printing and then different similarity thresholds for understanding the extent of copy/paste and reuse in them. We also provided an extensive analysis with different game engines and attempted to find out interesting insights. Furthermore, we provide an extensive comparison with open source non-games systems for studying varying nature and extent of reusability between them.

The most closely related work to our empirical study is the work of Roy and Cordy [136]. They conducted an empirical study on near-miss clones reusability in open source software and that we use kind of similar metrics. However, in their study there was no domain-specific clone analysis, and they did not use any game open source, whereas our focus is on games and game engines and then provide the comparative analysis with non-game open source software.

All detection tools perform code preprocessing before detection clones. Type of processing (Code transformation) depends on the underlying detection algorithm and data representation. In text-based detection, prepossessing includes the removal of whitespaces and pretty-printing [138, 19, 76, 158]. Token-based clone detection tools apply more processing and filtering. For example, Li et al. [108] apply keywords filtering, other keywords renaming, and data types and identifiers normalization. AST-based techniques [74, 48] include a significant transformation of source code into a tree structure. Despite all code clone detection tools use code transformation, there is no empirical study performed to investigate how much filtering and/or normalization should be done to achieve the best recall and precision. However, Deissenboeck et al. [42] referred that applying large filtering and normalization would produce a lot of false positive and reducing the filtering and normalization will miss a lot of clones.

3.7 Threats to Validity

One of the threats of our study would be the clone detector used for studying the copy/paste and reuse. We have used one of the state of the art clone detectors, NiCad which has been shown to give results with high precision and recall [139] not only for different types of clones but also for studying the copy/paste reusability for different similarity levels [136]. Another threat would be the limited number of samples. However, to the best of our knowledge this is the largest study on cloning reusability in games and that we further compared with non-games software. While we used NiCad with its pretty-printing and similarity thresholds, there is no guarantee that these returned clones are in fact the right ones for software maintenance activities such as refactoring. However, our target was just to find out the extent of reusability and not to find out clones that could be used for any maintenance activities.

3.8 Summary

With the fast growth of open source systems in different software domains, including game domains, there has been a need to analyze and compare the quality, maintainability, cloning status, and refactoring for each

domain. In this paper, we perform an empirical study to analyze the cloning status in and across open source games and compared the results with other open source non-games. We analyzed clones in more than 32 games open source of three different languages.

The findings of our study show that there is a considerable variation of cloning status (density, association with file, and localization) between games open source of different programming languages. On the other hand, the variation of cloning level between open source games of the same language is negligible. When comparing game open source to non-game open source, games have fewer clones than non-game open source and their clones are more localized. Finally, game open source domain tends to have less cross-game clones comparing to other open source systems. However, a lot of clones exist across recreated games. In such cases, code clone detection tool could be used to identify recreated (cloned) games.

Our results show that applying code transformation to the source code enables the detecting of more Type-2, Type-3, and Type-4 clones. On the other hand, some transformations that include filtering and abstraction could degrade the detection accuracy, and the researcher should be careful in applying these transformations. However, we believe that applying more code transformation, to some extent, would contribute to the detection of Type-4 clones. As the byte code and intermediate code is a well-normalized form of source code, we extend our research to investigate the ability to use the byte code and intermediate language to detect code clones, specifically semantic clones.

Chapter 4

Detecting Clones across Microsoft .NET Programming Languages

The Microsoft .NET framework and its language family focus on multi-language development to support interoperability across several programming languages. The framework allows for the development of similar applications in different languages through the reuse of core libraries. As a result of such a multi-language development, the identification and traceability of similar code fragments (clones) becomes a key challenge. In this paper, we present a clone detection approach for the .NET language family. The approach is based on the Common Intermediate Language, which is generated by the .NET compiler for the different languages within the .NET framework. In order to achieve an acceptable recall while maintaining the precision of our detection approach, we define a set of filtering processes to reduce noise in the raw data. We show that these filters are essential for Intermediate Language-based clone detection, without significantly affecting the precision of the detection approach. Finally, we study the quantitative and qualitative performance aspects of our clone detection approach. We evaluate the number of reported candidate clone-pairs, as well as the precision and recall (using manual validation) for several open source cross-language systems, to show the effectiveness of our proposed approach.

4.1 Introduction

Large software systems contain 10-15% of duplicated code which is also referred to as code clone [86]. Clones occur due to several reasons, such as software developers intentionally practice cloning to save time during software development, especially when reusing complete functional units. On the other hand, developers also unintentionally produce clones by re-implementing similar functionality (code fragments) that already exists in the same system or another existing system.

In general, code clones are considered harmful and can reduce the quality of software [83, 19]. For example, when a modification is performed to a cloned fragment, all other instances of this fragment may require the same modification. This task duplication requires not only additional maintenance effort but also results often in additional cost for the software project. On the other hand, not all clones are considered harmful. Some cloning patterns can benefit development and maintenance [86] and are therefore sometimes even considered as essential or unavoidable [163].

As software systems become larger, more complex, and are being developed using more than one programming language, clone management becomes an essential part of the maintenance process. One approach to clone management is the use of clone detection tools, which discover the presence or absence of clones in a software system. While single language clone detection has been widely addressed and matured [140], only limited tool support exists for clone detection in multi-language/cross-language software development [129, 128, 127, 151, 45].

More recently, there has been an ongoing trend towards multi-language software development to take advantage of different programming languages [95], specifically in the .NET context. For multi-language development, two key usage scenarios can be distinguished: (1) combining different programming languages within a single, often large and complex system, and (2) use of several languages for re-implementation of a current system to support new client, application, or due to non-technical reasons (e.g. [4]). As a result, the ability to detect and manage similar code reuse patterns that might exist in these multiple languages systems becomes essential.

Over the last decade, a variety of detection tools have been introduced [24, 140] which are typically based on parsing Abstract Syntax Trees (AST) or dependency graphs. These techniques use different matching approaches such as: string (token) similarity, vector similarity, sub-graph isomorphism or frequent set [24]. In addition, some clone detection tools do not rely on source code. They [20, 96] instead use bytecode or intermediate representations as their input.

While many clone detection tools are capable of supporting different programming languages, they lack actual cross-language support during detection time. Consequently, these tools only detect clones in one program language at the time, and do not detect clones that span over multiple programming languages. Few studies have been conducted on multi-language clone detection. For example, Kraft et al. [96] used unified representation for the .NET languages to detect clones between VB .NET and C# at source code level. Unlike early efforts, in this paper we focus on detecting clones across all Microsoft .NET programming languages (C#, C++ .NET, Visual Basic .NET, J# and F#) using Microsoft's Common Intermediate Language (CIL), as an intermediate representation of the disassembled binary .NET content.

In this chapter, we first highlight the importance of cross-language clone detection for Intermediate Languages. Second, since CIL is a low-level human-readable language, one has to deal with a larger amount of code which leads to additional variations and therefore poses new detection challenges. We established several preprocessing steps for the CIL code both to optimize it for clone detection and to reduce its content dissimilarity. Applying these different filtering and optimizations techniques allows us to improve the overall accuracy and performance of the clone detection. As a result our research addresses several fundamental research questions which are listed below.

- *RQ1:* Are "the selected filters" useful? In this experiment we observe how much it is likely that the filtering approach contributes to the true positive ratio.
- *RQ2:* Is our "clone detection approach" able to detect cross-language clones on .NET using Intermediate

Language?

- *RQ3:* How successful is the "clone detection approach" in terms of precision and recall?

We conducted several case studies on four datasets created from open source software systems written in C#, J#, and VB.NET. We used one of the datasets as our oracle for objective recall measurement. We applied three clone detection algorithms to avoid algorithm-dependent observation as much as possible. We manually investigated 2K clone-pairs (randomly selected) to measure our approach's precision and recall. Finally, we observed that our approach is able to detect high quality cross-language clones successfully at method level granularity using Intermediate Language.

This chapter is based upon the manuscript "Detecting Clones across Microsoft .NET Programming Languages" [11]. It was published by myself, Iman Keivanloo, Chanchal K.Roy and Juergen Rilling at the 19th Working Conference on Reverse Engineering (WCRE'12). I was the lead author of the work, and the other authors have taken the supervision role for the research. The publication has been modified and re-formatted to better fit in the context of the thesis.

The remainder of the chapter is organized as follows. In Sections 4.2 and 4.3 we provide the motivation and background about CIL and adopted clone detection tools and algorithms. Section 4.4 describes our proposed process and the filter set details. Section 4.5 studies the necessity of using filters (RQ1). We investigate the potential filters' effect on precision (RQ2) in Section 4.6. Finally, in Sections 4.7, 4.8 and 4.9 performance evaluation (RQ3 and RQ4), related work, and paper summary are presented.

4.2 Motivation- the Necessity for Unified Representation

In order to evaluate the necessity of using a unified representation (e.g., CIL or Kraft et al. [96] approach) as an intermediate representation for cross-language clone detection, we conducted some case studies. The objective of these studies is to establish a comparison between selected unified source code representation (CIL in the context of our research) and source code-based clone detection for cross-language clone detection.

For the case study, we adopt a language-independent comparison engine (the clone detection tool [138]). Second, we feed source code from different languages to the comparison engine. Finally, we repeated the clone detection process on the corresponding CIL content. We analyze the performance of the cross-language clone detection, by replacing CIL with the actual source code written in different programming languages. We conducted a quantitative study to evaluate and compare the detection results. For the study we detect cross-language clones in the different Mono compiler [7] versions (implemented either in C# or VB.NET), as well as the ASXGUI [1] C# and VB.NET versions. We then analyzed the reported clone clusters for both CIL and source code (Table 4.1).

Compared to source code, using CIL more cloned code fragments can be detected. Second, our manual validation of the reported clones showed that many of the missed clones at the source code level were actually near-miss clones, with these clones containing more than one line differences. In conclusion, CIL based clone

Table 4.1: Comparison between clone detected using CIL or source code as input

Dataset	Input Data Type			
	CIL		Source Code	
	# Clone Classes	# Clone Fragments	# Clone Classes	# Clone Fragments
ASXGUI	9	393	69	261
Mono	37	4373	369	1523

clusters always contained a larger number of code fragments with low cohesion. However, the overall recall and precision at the clone-pair level (not clone class) is higher compared to a source code approach and therefore shows that an intermediate language improves clone detection in a cross-language setting.

4.3 Background

In this section, first, we provide an overview of Microsoft's .NET Common Intermediate Language (CIL). Second, we review the code clone concept and the comparison algorithms used for this study.

4.3.1 Common Intermediate Language

The .NET Framework is a software development platform developed by Microsoft that runs primarily on Windows. It consists of several components such as, (1) a comprehensive library of commonly used functionalities, (2) a runt-time management environment (language independent), and (3) a set of programming languages. Contrary to Java, which targets application development using one language on several platforms, .NET aims for multi-language development on a single platform. It provides language interoperability, with each program module being able to use code written in the other languages.

The source code of various .NET programming languages (C#, Visual Basic .NET, Visual C++ .NET, J#, F# etc.) is compiled into the CIL. When the .NET managed code is compiled, the compiler first converts it into Common Intermediate Language (CIL), a machine independent intermediate language, before compiling it into .NET portable executable (PE). Visual Studio SDK includes a disassembler "ildasm.exe" that takes the Portable Executable (PE) file(s) and generates the CIL in human readable formats such as plain text (e.g. Figure 4.1–Column #2).

CIL is an object-oriented, stack-based like assembly language. It is also referred to as Microsoft Intermediate Language (MSIL) or Intermediate Language (IL). CIL is a platform independent language that can be executed in any environment that supports the Common Language Infrastructure. CIL itself is also a .NET programming language, which can be in combination with the CIL .Net compiler "ilasm.exe" also be used directly to develop applications in CIL.

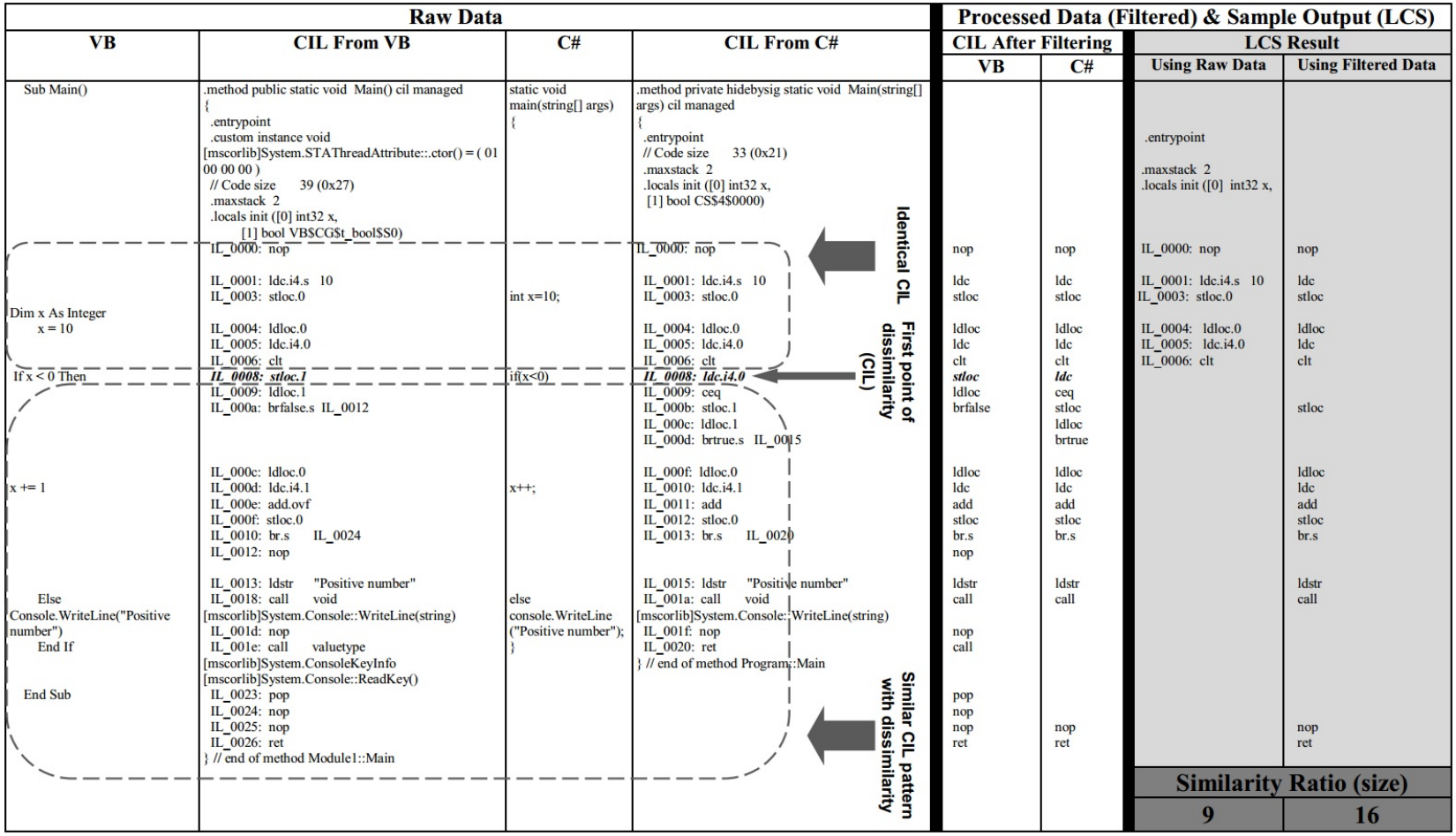


Figure 4.1: First Part (Left Section): A C#/VB methods including their corresponding CIL – The example shows the challenges in clone detection using Intermediate Language (e.g., larger size, unexpected dissimilarities in CIL). Second Part (Right Section): An example to show how our filtering proposal contributes to the clone detection by improving the similarity (e.g., LCS) between fragments.

Examples and Challenges. Being a lower level representation, CIL code size tends to be much larger than traditional high-level source code. Figure 4.1 (the first two columns) shows a comparison between a VB code fragment (a small VB method), and its corresponding CIL representation. In this example the method body with five lines of code has been transformed to more than twenty lines of code in CIL. This creates an additional challenge, making clone detection on binary rather different from source code.

Nevertheless, given this common representation of code fragments written in different programming languages provides the ability to use CIL for clone detection across .NET languages. However, a key challenge is the fact that it is possible to have some dissimilarity at CIL level, even in cases of semantically identical source code fragments (written in different .Net languages). The first four columns of Figure 4.1 (the Raw Data section) provide an example for such dissimilarities. Both the VB and C# methods implement the same program following similar coding pattern and structure as much as possible. However, when we compare the CIL pairs, there are three key sections clearly distinguishable: (1) identical CIL content which is marked by the first dashed area, (2) the first point of dissimilarity which is flagged by the italic font style, and (3) the rest of the content marked by the second dashed box that covers CIL content with considerable dissimilarity. In general, this example highlights the key challenge in binary clone detection, the possibility of facing dissimilarity by exploiting .NET Intermediate Language even for semantically (and almost syntactically) identical fragments in cross-language context.

4.3.2 Cross-Language Clones

Two code fragments that share some degree of similarity are typically considered a clone pair. Based on their actual similarity, clone pairs can be categorized [140, 24] as Type-1, Type-2, Type-3, and Type-4 clones. Type-1 clones are exact copies of each other, except for possible differences in whitespaces, layouts and comments. Type-2 clones are syntactically identical fragments except for variations in identifiers, literals, data types, whitespace, layouts and comments. Copied fragments (e.g., Type-1 and Type-2 clones) with further modifications such as additions, deletions and changes of statements are called Type-3 clones. Type-2 and Type-3 clones are also known as near-miss clones. Code fragments that perform the same computation (e.g., semantically similar) but implemented through different syntactic variations are called Type-4 clones. Note that all of these definitions were originally introduced for clone-pairs implemented in the same programming language. In our cross-language clone research these definitions are no longer applicable as-is, and have to be refined to meet our research context. For example, the VB and C# fragments in Figure 4.1 would be considered Type-1 clones in the cross-language clone detection since they are essentially performing the same task implemented in different programming languages.

4.3.3 Matching Algorithms

In our research we use SimHash [165], Longest Common Subsequence (LCS) [71], and Levenshtein Distance [106] algorithms to detect clone-pairs. Note that the first two techniques are adopted from SimCad [16] and

NiCad [17] respectively. Our primary research goal is to address clone detection challenges in the .NET Intermediate Language by providing solutions that are generally applicable and independent of a specific clone detection algorithm. We therefore selected three dominant algorithms for modeling edit distance in this chapter to avoid an algorithm specific solution.

The Longest Common Subsequence (LCS) [71] algorithm detects the longest common subsequence between two strings. For example, consider the following two sequences of characters.

S1 =AABBBCDABCDDAABD

S2 = DDABCCCDAABBBDAC

For the above example, the LCS among the S1 and S2 sequences is ABCDABDA. For our cross-language clone detection we applied LCS to the CIL instruction sequences in order to determine the LCS similarity of CIL code fragments (e.g., highlighted part in Figure 4.1). The LCS size is important since it is the similarity measure we use to decide if two fragments form a candidate clone-pair in our research. We used the LCS to measure the similarity between two sequences of tokens as:

$$LCS\_Sim(s1, s2) = \frac{LCS(s1, s2)}{\frac{size(s1) + size(s2)}{2}} \tag{4.1}$$

Levenshtein Distance (levDist), also called Edit Distance, is defined as the minimum number of insertions, deletions, and substitutions of characters required to transform one string into the other [106].Contrary to LCS (where the actual output is a string), *LEVDist* provides as output the dissimilarity between two string sequences as a single number value. We use *LevSim* (Equation 4.2) and its output is compared against a constant threshold value, to decide whether two fragments (i.e. S_1 and S_2) will be reported as candidate clone-pairs.

$$LevSim(s1, s2) = 1 - \frac{LevDist(s1, s2)}{max(size(s1), size(s2))} \tag{4.2}$$

SimHash-based Clone Detection, SimHash algorithm constitutes the core of SimCad [165]. It generates a 64-bit fingerprint, which we use to detect clones based on their fingerprint similarities. The algorithm uses Charikar's [26] hash function where the Hamming Distance is used as the crucial configuration parameter. The Hamming Distance represents the number of positions at which the corresponding bits are different between two fingerprints. A Hamming distance of zero corresponds to identical fingerprints and therefore also to Type-1 clones, while a Hamming distance larger than zero reflects near-miss clones.

4.4 Clone Detection Process (Across .NET Languages)

Figure 4.2 shows the overall processing steps for detecting clones across Visual Studio .NET programming languages. First, the .NET language source code is collected and compiled to obtain the .NET portable executable files. During the second processing step, the disassembling of corresponding executable files takes place through the Microsoft's Intermediate Language Disassembler (ildasm.exe). The disassembler generates the CIL code as plain text files. The CIL files are then parsed to extract all method/function bodies. In the next step, we apply our filters on the CIL code. The filters play a key role in our approach since they eliminate undesirable noise and improve the overall quality of the CIL files. After applying the filters on the CIL, we run the selected comparison algorithms to detect clone-pair.

As part of this clone detection phase, we use, (1) LCS, (2) Levenshtein Distance and (3) SimHash-based algorithms. Finally, a clone-pair report will be generated for both CIL and source code (by mapping the CIL clones to their corresponding source code).

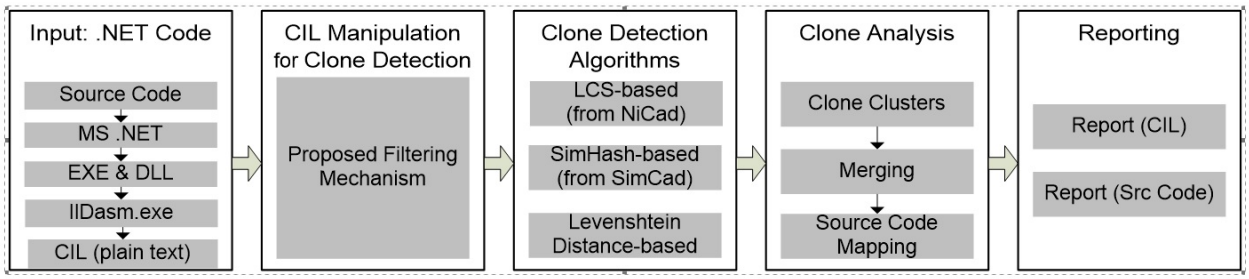


Figure 4.2: Schematic diagram for the proposed cross-language clone detection and result evaluation.

4.4.1 Filtering Out the Byte Code

Figure 4.1 shows an example of a code snippet written in C# and VB including their corresponding CIL representations. The key challenge (as discussed earlier): although both methods implement the same functionality and are compiled into the same Intermediate Language, still there is some dissimilarity in their intermediate representations, which can affect the clone detection process. We address this challenge by creating a set of cleaning and filtering steps for CIL to improve the performance of Type-1, Type-2, Type-3 and Type-4 clone detection in the CIL code. The filters are designed to improve the detection rate (i.e., recall) since the CIL data contains a significant amount of noise (e.g., reference numbers to string tables, which are compilation context dependent). Due to such noise in the CIL files, two semantically identical source code fragments might no longer be considered as highly similar at the CIL level (e.g., content similar VB and C# methods might have less than 50% similarity at the CIL level, see Figure 4.1).

4.4.2 Overview of the Proposed Filters

Prior to the actual filtering process a pre-processing step takes place that removes all directives (e.g., .method, .entrypoint, and .maxstack) that appear at the beginning of the CIL representation of a particular method. After this preprocessing step, the actual CIL code sequences are analyzed and filtered accordingly. In what follows we provide a summary of the eight filters we create (Table 4.2) and provide an example for each filter.

- **Filter 1:** Removal of the instruction address (IL_xxxx:) at the *begin* of each CIL instruction, eliminating dissimilarities due to application/environment specific variations.
- **Filter 2:** Removal of instruction address (IL_xxxx:) for *branching statement*. As part of this filtering step we cover all 33 branching statements. For example, `beq, beq.s, bge`.
- **Filter 3:** Removal of integer values that represent argument number in CIL. For example, `ldarg 3` is interpreted in CIL as load the argument number 3 onto the stack. Instructions included in this filter are: `starg, starg.s, ldrag, ldrag.s, ldrags, and ldraga.s`.
- **Filter 4:** This filter eliminates *constants* in the CIL code, e.g. `ldc.i4 num` which corresponds to a *Push num* of type int32 onto the stack as *int32*. Instructions covered by this filter are `ldc.i4, ldc.i8, ldc.r4, ldc.r8,` and `ldc.i4.s`.
- **Filter 5:** This filter removes all print literals in the CIL code, which are identified through `ldstr` statements.
- **Filter 6:** This filter removes all variable indexes like `stloc index`, which correspond to popping a value from stack into a local variable. Among the instructions removed by this filter are: `ldloc, ldloc.s, ldloca.s, stloc and stloc.s`.
- **Filter 7:** This filter removes some additional data types and constant integers such as `i4` from `ldc.i4. 1`. The complete command pushes 1 as an int32 onto the stack.
- **Filter 8:** Is not actually a new filter, it combines all seven filtering techniques mentioned above, including the preprocessing tasks in one single filter.

Table 4.2: Examples of CIL filters

	Before Filtering	After Filtering	Example Description
Filter 1	IL_0003: stloc.0	stloc.0	Where IL_0003 is the instruction address address
Filter 2	brtrue.s IL_0015	brtrue.s	The IL_0015 address of the branch destination destination
Filter 3	ldarg 3 starg 1	ldarg starg	The value 3&1 represent argument number number
Filter 4	ldc.i4.s 10	ldc.i4.s	10 is the number (pushed to the stack) stack)
Filter 5	ldstr "Positive number"	ldstr	"positive number" is the printed string constant constant
Filter 6	stloc 7	stloc	7 represents variable index index
Filter 7	ldc.i4.s 10	ldc	i4 represent the int32 data type in CIL and s for short for Short
Filter 8	IL_0011: add IL_0012: stloc.0 IL_0013: br.s IL_0020 IL_001a: call void [mscorlib]System.Console::WriteLine (string)	add stloc br call	Note that Filter 8 is just a nick name. Refer to the Filter 8 description section for more details

4.5 Filters' Contribution

As discussed earlier, we propose this set of filters to increase the recall (by reducing noise and dissimilarity in the CIL code) and to be able to detect other valuable clone types such as type-3 clones. For example in Figure 4.1, we were able to successfully increase the similarity ratio of the clone-pair from 9 (before filtering) to 16 (after applying filtering). In order to answer in more detail our first research question (RQ1), we conduct an experimental evaluation to determine how much our filtering approach actually contributes to the true positive ratio and its benefits to the overall detection process.

To answer this question, we defined a metric called *Filter Contribution* that measures the effectiveness of each filter. The underlying idea is to measure the similarity degree of candidate clone-pairs before and after applying different filters. The measurement will indicate how much a particular filter increases the similarity value between two fragments. Note that in the ideal case, we expect that a filter would increase the similarity values of true positive cases significantly more than the ones for false positive cases. Otherwise, a particular filter would not be useful to discriminate (with high confidence) against false positives. The Filter Contribution (*FltrCntrb*) function is defined in Equation 4.3, which is based on LCS-based similarity (Equation 4.1). S_i denotes the participant fragments in the clone-pair under investigation and F_x presents the filter function with x being the filter number.

$$FltrCntrb = LCS\_Sim(F_x(S1), F_x(S2)) - LCS\_Sim(S1, S2) \tag{4.3}$$

The challenging part of this experiment was to identify proper input data, due to external constraints caused by the availability of .NET source code on the Internet. More specifically, the challenge was that

we had to identify software systems written in more than one .NET language. Fortunately, the iText.NET package [3] (note that it is different from iText project [4]), met our input constraint. Although iText.NET is originally written in J#, it includes C#, VB, and J# methods for its 25 major use cases. Therefore, our first dataset (a.k.a. Cloned Fragments Dataset) contains 25 clone classes, with each clone class having three clone fragments, with code fragments being implemented using three different .NET programming languages (C#, VB and J#). Note that we mutually created three true positive clone pairs by following the VB-C#, VB-J#, and C#-J# patterns for each use case using the iText.NET API usage code [3] (Example Code Section). This approach resulted in 75 distinct clone pairs (i.e., actual true positive clone-pairs). The second dataset (a.k.a., Non-cloned Fragments Dataset) contains 25 non-clone classes and 75 false positive clone-pair candidates created in the same manner as clone classes. We want both tagged datasets to be able to help answer RQ1 (i.e. whether filtering has any positive/negative effects on cross-language clone detection).

We then measured the *Filter Contribution* value for each filter when applied on both datasets. The result for each data set is shown in Figure 4.3 and Figure 4.4. We exclude Filter 8, since this filter does not introduce a new threat (i.e., no negative effect) to our clone detection approach, since this filter only engages all other filters. In most cases, the filters increased the similarity up to 0.2 (max) for non-cloned pairs while improving the similarity of cloned pairs by at least 0.3. This result supports our research hypothesis that filtering increases the similarity values for true positive cases (the cloned dataset) with a higher ratio than the false positive cases (the non-cloned dataset). Comparing the results of Filter 8 between Figure 4.3 and Figure 4.4, it is observable that the answer to RQ1 is positive since the overall contribution of the filters improves the similarity degree on actual cloned pairs much more than for non-cloned fragments (non-cloned pairs less than 0.5, while for the majority of cloned pairs the similarity increases between 0.5 and 0.8).

To support our claim, we conducted another case study on the same dataset to determine if our filters can be used to identify an appropriate similarity threshold. Figure 4.5 summarizes the findings, showing that before applying our filters, there was no clear distinction between similarity values of actual clone-pairs (true positives) and false positives. Therefore it is impossible to determine an adequate threshold that allows separating actual clones from false positives. In contrast, Figure 4.5 shows that filters address this problem by increasing the distance between the two groups (tagged on the right side of Figure 4.5). For example, using our filters, a threshold from 0.4 to 0.55 can separate true positives from false positives with high confidence. Our analysis therefore supports the usefulness and necessity of the proposed filter set (RQ1) for cross-language clone detection on CIL.

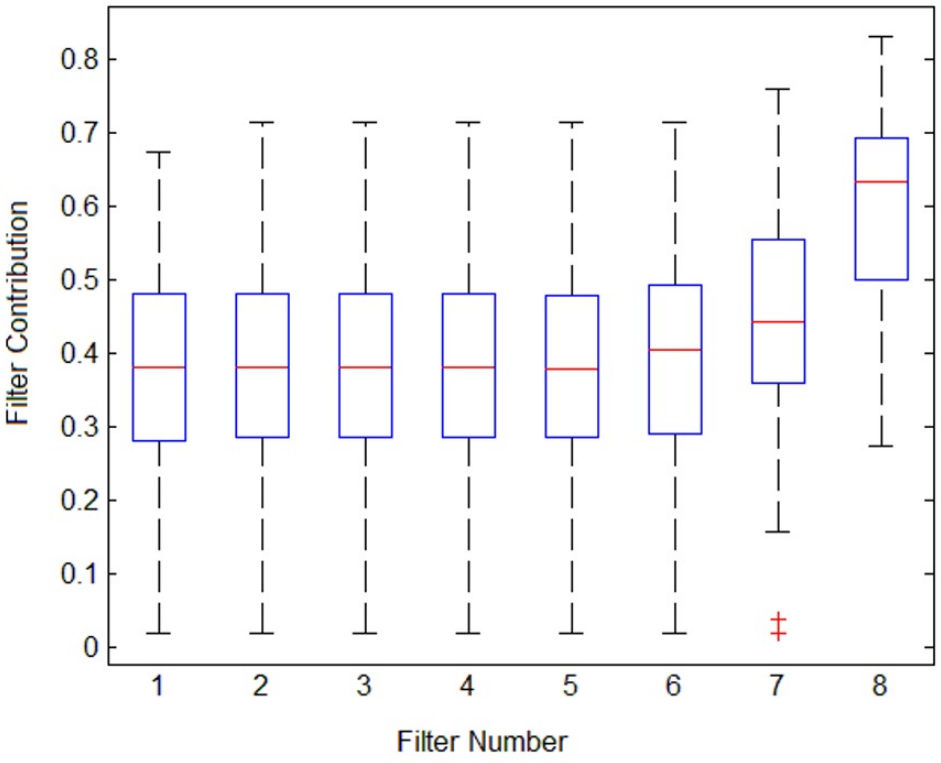


Figure 4.3: Filtering Effects on the Cloned Dataset

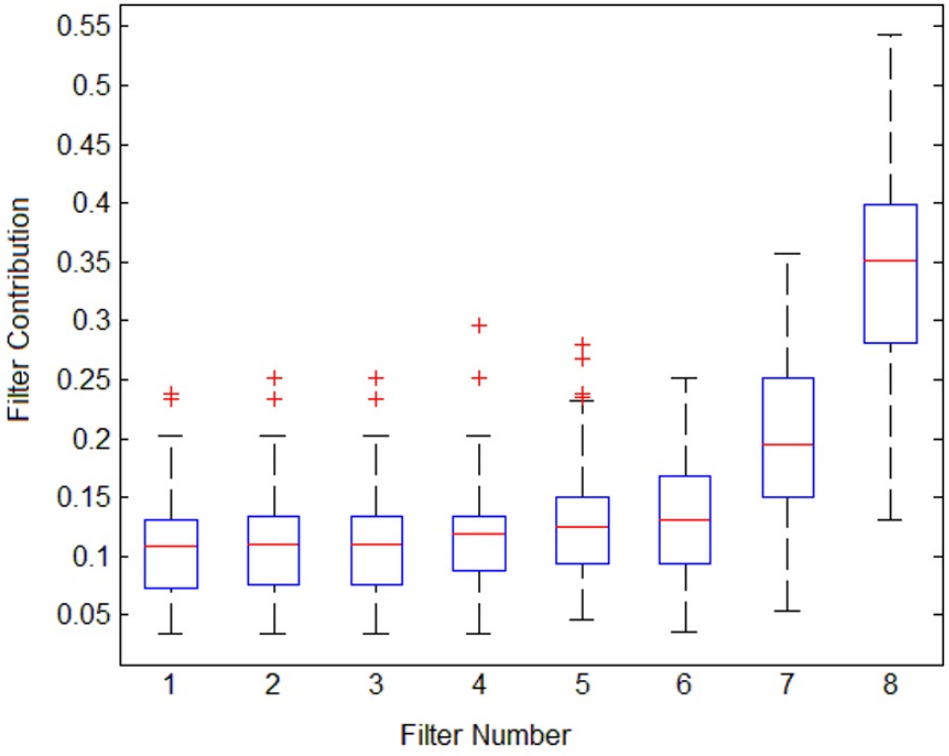


Figure 4.4: Filtering Effects on the Non-cloned Dataset

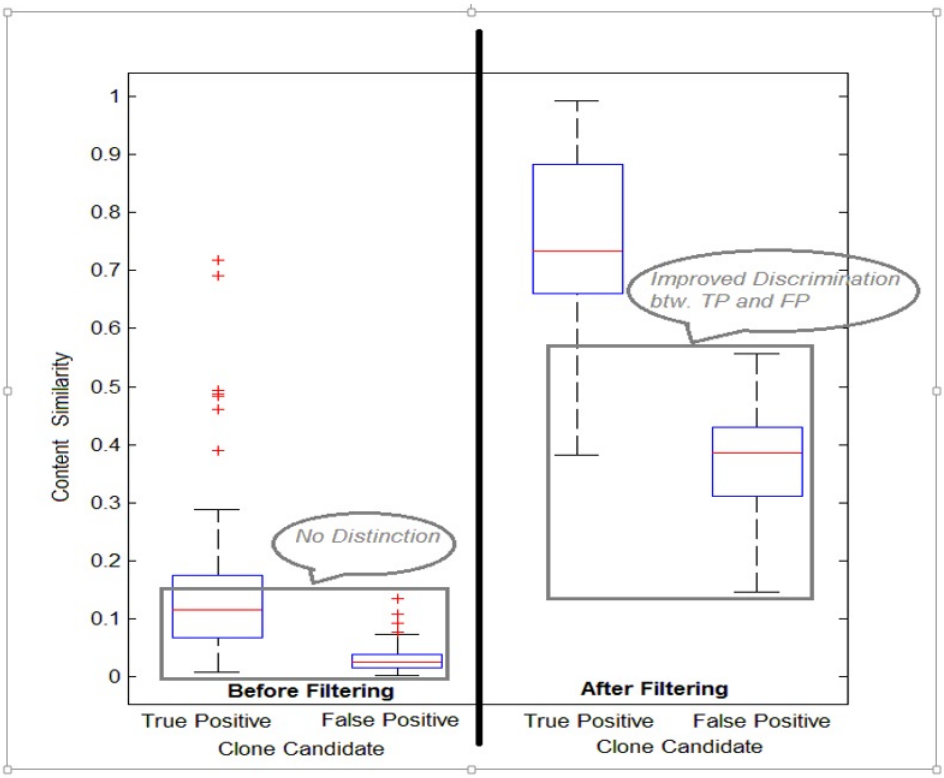


Figure 4.5: Filters' contribution to discriminate between actual clone pairs and non-cloned pairs

4.6 Does Filtering Make Actual Clone-Pairs and Non-Cloned Pairs Similar?

A major threat to any filter-based approach is the loss of precision by filtering out essential data. As a result, excessive or improper loss of data (due to filtering) can lead to a situation where *non-answers* and *actual answers* become similar to the decision making algorithm, which eventually leads to an increase in the false positive ratio.

In this research, we are interested in observing the similarity between actual clones and non-cloned fragments after filtering (to answer RQ2). Apparently, we prefer to have fewer similar entities when we compare members of actual clone-pairs with other paired fragments (i.e. non-cloned fragments) to support the applicability of our filtering approach.

In order to observe the similarity/dissimilarity of code fragments we adopt the Chernoff face [30] visualization approach. The Chenoff face visualization represents data as glyphs similar to human faces while each dimension is being mapped to a specific feature of the face (e.g., the Filter #5 similarity contribution value determines the distance between eyes for each clone-pair). We application of glyphs in software visualization is a well-established research area (e.g. [31]).

For our controlled experiment, we produced seven face features for each pair by calculating *Filter Contribution* on all seven filters separately. That is, each pair can be modeled using a vector in a multi-dimensional space (in our case, seven dimensions). Based on this assumption, two sub research questions arose (RQ2 breakdown): (RQ2-a) "do filters make actual clone-pairs similar to non-cloned pairs?". In other words, "Does the filtering approach mislead the clone detection approach to report a false positive as clone-pair?" or "is filtering a major threat to false positive ratio of cross-language clone detection using CIL?", and (RQ2-b) "Is filtering neutral to the participating programming languages of clone-pairs (in cross-language clone detection context)?".

Figure 4.6(a) and Figure 4.6(b) show the result for two datasets (similar data as in Section 4.5). By comparing the faces between Figure 4.6(a) and Figure 4.6(b), it is possible to answer the RQ2-a: filtering does not make non-cloned pairs similar to actual clones. Therefore, filtering becomes not a major threat for the precision in our research. For example, we can observe that there are more distorted and super tiny faces (i.e., pairs - e.g. the second face of Figure 4.6(a)) available in Figure 4.6(a) which contains non-cloned pairs than Figure 4.6(b). The issue can be attributed to Filter 1, 2, and 5 since they are mapped to: (1) the face size, (2) distance between forehead and jaw, and (3) distance between eyes respectively. Therefore, it is also possible to intuitively observe that Filter 1, 2, and 5 (including Filter 7 observed in Figure 4.4) play the major role in characterization of true positives.

To answer RQ2-b, we categorized the clone pairs based on the programming language. Figure 4.6(c) and Figure 4.6(d) illustrate the result. For example, the top category in Figure 4.6(c) contains all pairs where the first fragment is written in VB and the other fragment is in C#. As it is obvious C#-J# pairs

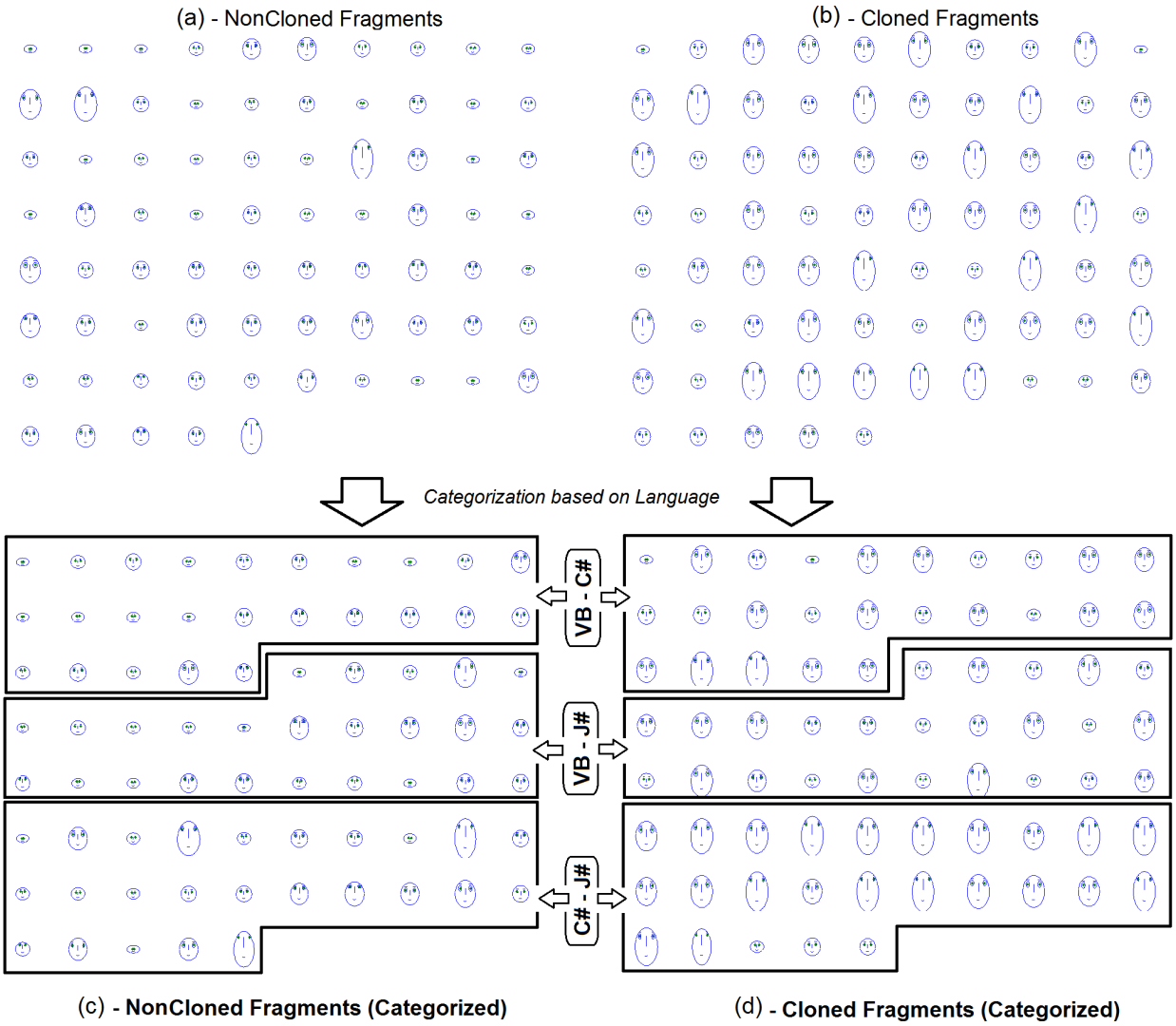


Figure 4.6: Filter Contribution data are mapped to multi-dimensional space to investigate the importance of filters and language dependency using glyph visualization. Note that numbers are used for referencing purposes in this figure (i.e. there is no ordering relation between faces etc. specifically between faces in the left and right hand sides – in short each face represents just one candidate clone-pair).

in Figure 4.6(d) (cloned pairs) are different. That is most of the faces are not round shaped comparing to the two other groups in Figure 4.6(d). The same pattern is observable in Figure 4.6(c) C#-J# category which contains mostly non-rounded shaped faces. Therefore, we can confirm RQ2-b, filters are independent of the programming language. This observation can be attributed to the resemblance between C# and J# languages and their history where the IL content becomes highly similar where we filter out the line numbers (i.e., Filter 1 and 2).

4.7 Evaluation

In this section, we present the evaluation results from our cross-language clone detection using Intermediate Language on four .NET systems. We analyze the clone detection results from a qualitative and quantitative perspective, using three edit distance methods (LCS, LEV, SimHash) to answer RQ2 and RQ3. Note, all datasets used for the evaluation include at least two .NET languages.

The first dataset contains two versions of the ASXGUI [1] open source GUI encoder. The dataset contains Version 3.0 and 2.5 since its current version (Ver. 3.0) is developed in C#, while the earlier implementation was based on VB.NET. ASXGUI Ver. 2.5 consists of 47 VB.NET files with a total of 32594 lines of source code and 303 functions. ASXGUI v 3.0 on the other hand, consists of 19 C# files with 2088 lines of source code, and 78 functions. The combined number of files being analyzed is 66 files with a total of 34682 lines of source code. The noticeable difference in project metrics (e.g., LOC) can be attributed to the (1) dissimilarities in the programming languages, and (2) re-engineering and refactoring tasks.

The second dataset is based on the C# and VB.NET compilers from Mono [7], version 2.10. The C# compiler consists of 57 C# files and the VB.NET compiler of 375 files, with a combined number of 432 files and 4998 functions.

The other two systems used in the study are two PDF libraries called iText and iText.NET. While their project names are similar, both projects are completely independent from each other. We created our third dataset from the iText (C# branch) and iText.NET (J#) source code. The dataset contains more than 600K LOC and 2.5K files.

We used part of iText.NET library to create our last dataset. This dataset contains source code related iText.NET API usage written in three languages (C#, J#, and VB.NET). This feature makes the dataset an important resource for our study since it allowed us to create a small (75 clone pairs) but controlled dataset (i.e., all actual clones are aligned, tagged and known in the cross-language), creating a unique oracle for further analysis. We use this oracle to obtain precise recall and precision measures, since the number of actual clones is known. This is contrast to the other datasets, where recall and precision measure cannot be computed as precisely, since the actual number of clone-pairs is unknown.

4.7.1 Quantitative Evaluation

Figure 4.7 shows the total number of detected candidate clone-pairs from the filter datasets using the three selected distance measure algorithms. The results from this experiment can be summarized as follows: (1) it is possible to detect numerous candidate clone-pairs even for cross-language case regardless of the underlying algorithm, (2) no candidate clone-pair is detected for cross-language using 1.0 as the Similarity Factor (i.e., the decision making threshold), which would only report clone-pairs with complete identical content. Therefore, even using *filtering* on highly similar cross-language clone-pairs (e.g., Figure 4.1), some dissimilarities will have to be handled by the clone detection approach. However, this is not the case for single language clone detection (shown in Figure 4.7), (3) for all dataset, we can observe a major decrease in the number of candidates when the threshold value is set to a range between 0.6 and 0.8 (marked by ovals). Therefore, we can conclude that the detected range can be recommended to the end-user to provide an acceptable recall and precision (the supporting argument is given in the qualitative evaluation section). The only exception is SimHash, which reports for the same thresholds a lower number of candidate clone-pairs. This is due to the fact that SimHash uses a different threshold schema compared to the other distance measure algorithms.

4.7.2 Quantitative Evaluation

Since a noticeable number of candidate clone-pairs have been detected (Figure 4.7) using the *filtering* approach, it is necessary to evaluate the quality of our approach, by manually validating the results (RQ3). We manually examine candidate clone-pairs to determine whether they are true or false positives. We investigated three distinct thresholds from the selected range discussed in the previous section, which we refer to as Extreme (threshold = 0.6), High (threshold= 0.7), and Normal (threshold = 0.8) configurations. The objective is to observe the best and worst achievable true positive ratio using the Normal, High and Extreme configurations. We also applied random sampling since the total number of candidates even for the chosen thresholds was considerably large (e.g., 10K clone-pair). Finally, we evaluated 2K candidate clone-pairs which were selected randomly.

(i) Challenges in Quality Assessment for Cross Language Clone Detection

Quality evaluation is inherently challenging in our research since there is no clear agreement on what constitutes true positives (TP) and the various clone types definitions. Therefore, we applied in our qualitative evaluation the following approach: (1) since it is possible to easily locate with confidence false positives among candidate clone-pairs, we first tag all false positives; (2) we assume the rest as true positive. However, in order to provide a more in-depth quality assessment, we also analyze the quality of the reported true positives. One of the interesting examples which we identified in the ASXGUI dataset is the true positive shown in Figure 4.8. For this example it easy to select an appropriate corresponding clone type based on the existing defacto clone definitions (e.g. [140, 24]). Regardless of dissimilarities introduced by different languages (e.g., VB.NET vs. C#), it is obvious that: (1) both methods in this example are implementing the same functionality, and

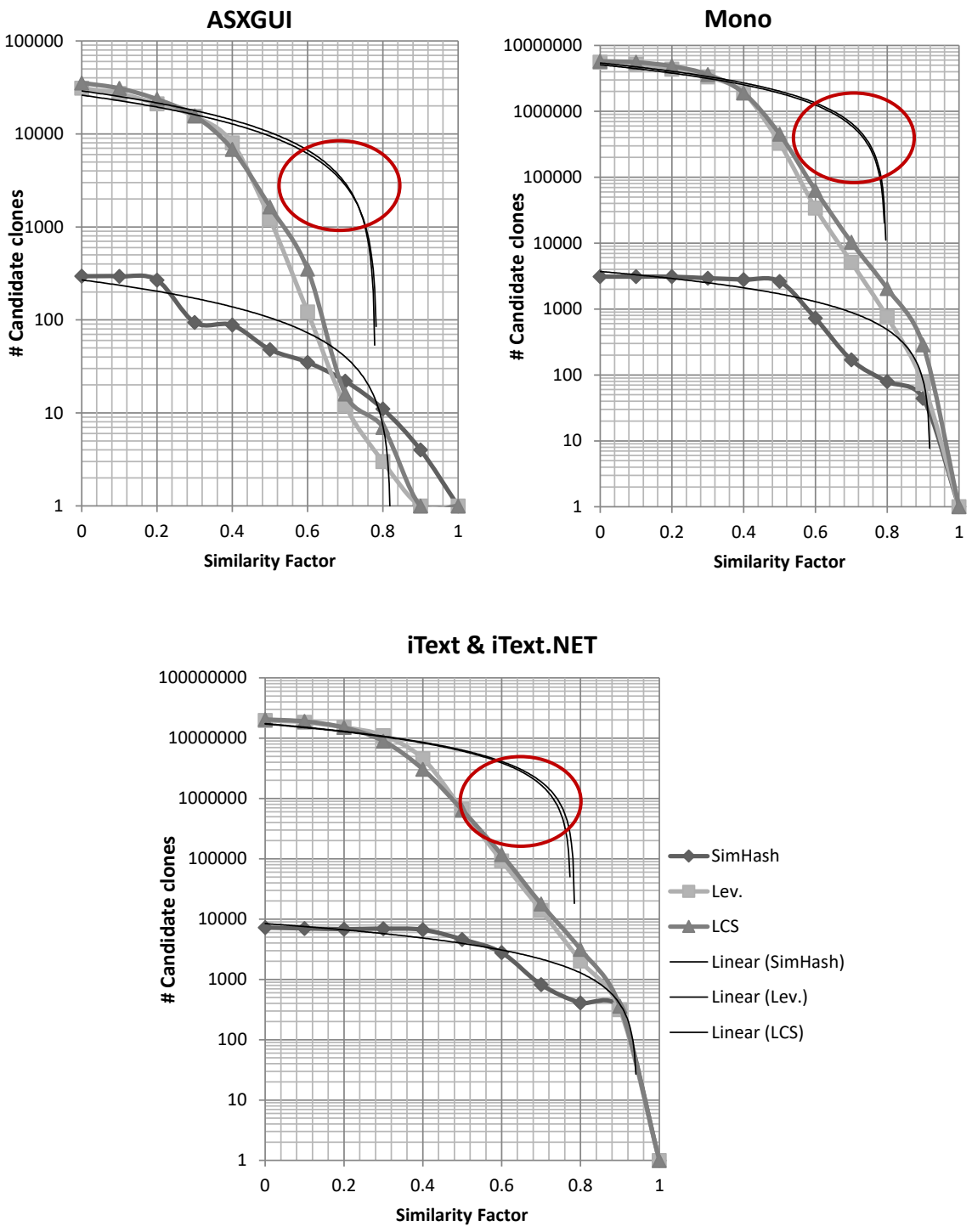


Figure 4.7: Number of clone-pair candidates per (1) dataset, and (2) clone detection algorithm. Note that the Similarity Factor varies between 0 and 1. For example, 1 is the strictest threshold which leads to the detection of only exact content. The thin black lines show the linear trend for the corresponding case study. However they appear as curved lines since the horizontal axis is logarithmic on purpose. Following this approach, it is possible to observe the major drop area (threshold) in number of detected candidate clone-pair which is between 0.6 and 0.8 for almost all datasets and algorithms (with few exceptions).

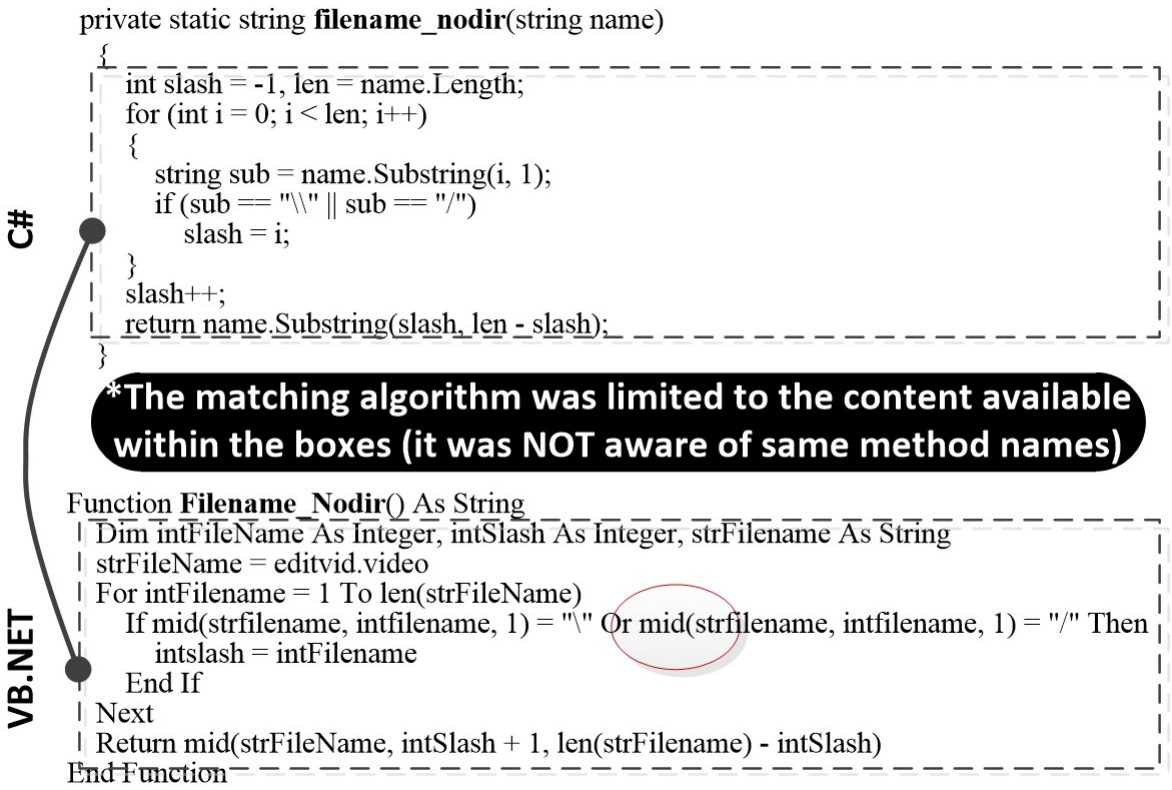


Figure 4.8: An example of two strongly similar clone-pair detected by our approach from ASXGUI. Even in case of major dissimilarity such as occurrence of mid method in the VB section for several times, still our approach successfully detected the clone-pair.

(2) most importantly they are following the same algorithm. Therefore, although we cannot select the clone type, we consider such clone-pair candidates as *strong true positive* in our *qualitative evaluation*. Therefore, we consider all clone pairs similar to this example as *strong* TPs and label them with E, and the remaining TPs are being labeled as S.

(ii) Quality Evaluation Result

Figure 4.9 reviews the findings of our quality evaluation from manually assessing 2K candidate clone-pairs (answering RQ3). In general, using the *Normal* threshold all candidate clone-pairs that were reported are true positive (100% TP). The quality decreases with less restrictive thresholds. For example using SimHash and the Extreme threshold, the reported TP reduces to 40%. The optimum, considering the trade-off between precision and recall, was achieved using Levenshtein Distance-based comparison with the High threshold (80% TP). Nevertheless, this result is not 100% precise (threats to validity) due to the sampling process and data dependency.

The other major aspect of our qualitative evaluation is the recall measurement (RQ3 recall section), which we calculated on our only available oracle (iText.NET API). In our evaluation, we observed a recall of 76% using High threshold between three languages (C#, J#, and VB.NET). Note, we did not compute the recall for the other datasets, due to the lack of an objective assessment of what constitutes an actual TP for these

datasets and consequently, would make any recall computation for these datasets prone to subjectivity.

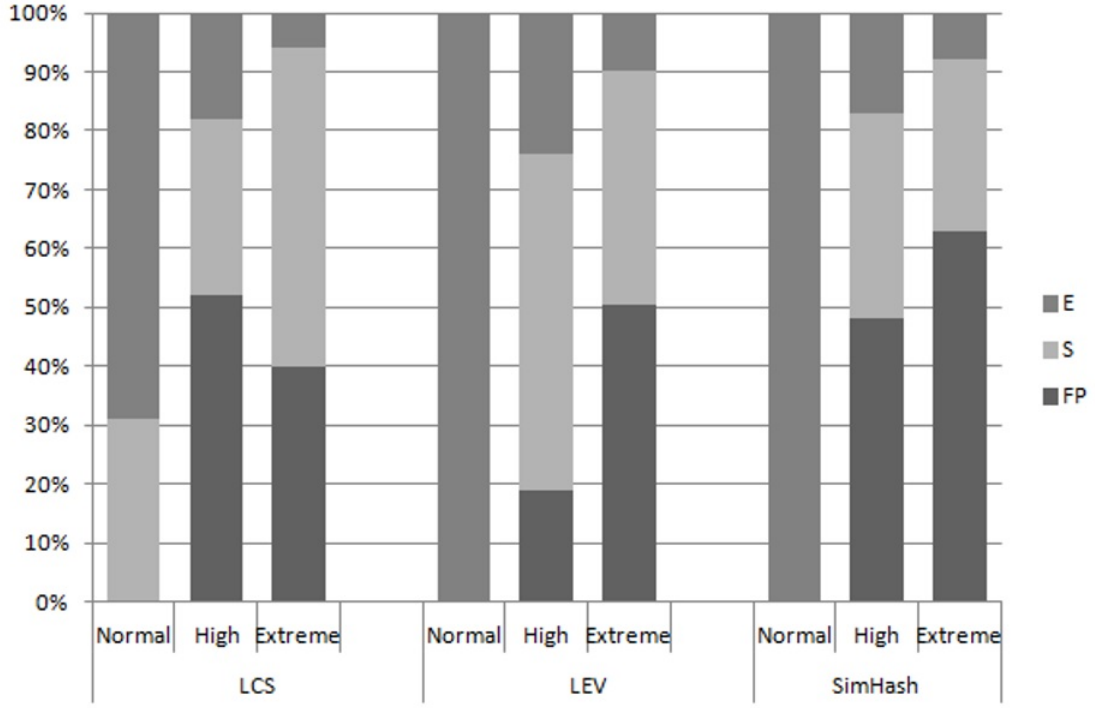


Figure 4.9: Result of true/false positive (within candidate clone-pairs) evaluation using manual analysis. Normal, High, and Extreme stand for corresponding thresholds from the selected range in the Quantitative Evaluation Section.

4.8 Related Work

While there are numerous well-established clone detection tools available to support different programming languages [140, 24], there exist only very limited research related to (1) *cross language*, or (2) *binary-level* clone detection. To the best of our knowledge, C2D2 [96] is the only tool capable of detecting cross-language clones. It uses NRefactory Library to generate the Unified CodeDOM graph for both C# and VB.NET. A string is generated by traversing this graph and targeted to string matching algorithm.

There are a few but diverse approaches on Intermediate Language-based clone detection (focusing on single language clone detection, mostly Java). One of the first studies on Intermediate Language clone detection is by Baker [20]. After some preprocessing (e.g., remapping offsets), she uses three comparison techniques (e.g., Diff [70]) to find similar fragments. Davis and Godfrey [40] use the disassembler for both Java and C/C++ to detect clones in single language. They provide a public framework for pretty-printing of disassembled code which constitutes the baseline of the clone detection phase. The most interesting aspect of their research is the proposed search algorithm for content matching which has two greedy and hill-climbing analysis steps. Selim et al. use "Jimple"[5] to detect clones using an intermediate representation. The motivation [146] is to exploit

Jimple characteristics (comparing to stack-based Java native IL). Recently, Juricic [78] uses Intermediate Language code to detect plagiarism and similarities. The approach is based on Levenshtein Distance as the similarity measure to compare disassembled C# binary, and applies some primitive preprocessing techniques which are comparable to two of our filters. There are also some formal approaches, such as by Cuomo et al. [36] that transform Java bytecode to mathematical models for clone detection.

To the best of our knowledge, our study presents the first comprehensive research focusing on, (1) .NET clone detection, (2) across programming languages, and (3) using Intermediate Language. Moreover, we not only proposed the approach, but also evaluated its major potential threats using diverse statistical and intuitive analyses. Finally, we evaluated its performance using manual validation to measure precision and recall. We observed a promising result in terms of both quantity and quality using three clone detection techniques for C#, J#, and VB.NET cross language clone-pairs.

4.9 Summary

With the globalization of the software industry and introduction of new programming languages, there has been an ongoing trend towards combining or re-implementing systems using different programming languages. This poses a new challenge for software comprehension, maintenance, clone management, and refactoring.

In this paper, we study a novel approach to detect cross-language clones in the Microsoft.NET Environment. In our research, we exploit CIL, an intermediate representation generated by .NET compiler from all .NET programming languages. We established a filter set (containing 7 filters) which are applied to CIL prior to the actual clone detection process to remove noise in the dataset and establish threshold values for the detection algorithms. Using face glyphs, we showed that the filters do not remove crucial data from CIL and therefore have no negative effects on clone detection.

Finally, we performed both qualitative and quantitative studies of clone detection approach on four datasets. We used three widely used edit-distance functions to allow for a generalization of our observations and study results. In our evaluation, we observed that it is possible to detect a reasonable number (quantity) of clone-pairs with acceptable precision (based on the configuration) and recall (quality).

The .NET intermediate code represents a normalized form of the source code. It represents the source code in a lower-level code. At the same time, it encompasses all the source code artifacts and relationships. In this study, we were able to detect cross language code clones by using the .NET CIL. We believe that the intermediate language has the ability to detect same language clones and semantic clones. In the next chapter (Chapter 5), we examine the capability of the CIL in detecting same language clones and semantic clones.

Chapter 5

Detecting Semantic Clones Using .NET Intermediate Language and Automatic Ontology

There are a great many code clone detectors available that target detecting different types of clones. While the majority of them are based on the syntax of the source code, and can detect syntactic clones accurately, only a small subset of them target semantic clones (Type-4). Furthermore, the accuracy of these tools for detecting semantic clones is either low or not up to the mark. In this chapter, we proposed a semantic clone detection tool that works at the byte code level. We first build a byte code ontology that describes the intermediate code, its structures, and relationships. Then, we use this ontology to detect clones at the byte code level. Our proposed technique is able to detect Type-4 (semantic) clones with higher accuracy than the state of the art. It also detects Type-1, Type-2 and Type-3 clones with very good accuracy. We extensively evaluated the proposed approach both by comparing with the state of the art clone detection tools (in particular those detect semantic clones) and by a mutation-injection based framework. For the Mutation-injection framework, we reviewed the definitions of semantic clones in the literature and proposed a comprehensive taxonomy of mutation operations that produce different types of semantic clones. We then used this framework to evaluate our tool and compare the results with a number of syntax and semantic clone detection tools. The results show that our technique is able to detect some gapped clones and semantic clones that are not detectable by state-of-the-art clone detectors.

5.1 Introduction

Identifying duplicate code in system software is important for code analysis and comprehension. Sometimes, developers are interested to know if there is existing similar code to the one they are developing in order to avoid creating duplicated code. Another example, when updates are needed to apply to the source code; it is important to identify the similar code fragments that might need to apply the same changes or even consider whether these clone fragments need to be refactored. Clone detection tools are used to identify such duplication or functionality. Clone detection tools became an integral part of IDEs. For example, in Microsoft Visual Studio 2012 and later versions, users can identify similar code fragments. For some other detection tools, a plugin is created that enables users to identify or search for code clones [43, 64].

Code clones could be syntactic similar or semantic similar. Syntactic clones are classified into clone Types-

1, Type-2, and Type-3 according to the degree of similarity [141]. Semantic clones refer to functionally similar codes that are not syntactically similar [141]. Detecting clones is usually done by transforming the source code into an intermediate representation (pretty-printed code, tokens, AST, PDG, or Byte code) and detecting the matching between new representations to identify potential clones. PDG-based approaches capture the flow and data dependencies in source code and could detect some of semantic clones [53]. The key challenge would be to find a technique that is capable of detecting clones (syntactic and semantic), that are not detectable by most state of the art clone detectors with acceptable accuracy.

Many published clone detection tools have perfect accuracy in detecting syntactic clones [140, 24]. However, some tools are better than others in detecting some types of clones. On the other hand, current semantic clone detectors have limited capability of detecting semantic clones [167] accurately. However, most of these detectors are good in detecting a sub-type of semantic clones. For example, [149] is good for code relatives, [50] is good for simions, and [87] is good for pattern-similar code.

Semantic clone detection is considered the main challenge in the area of clone detection. A number of techniques have been proposed in the literature to detect such clones [65, 87, 47, 53, 100, 149, 88, 178, 143, 125, 25]. Most of these are not based on a solid definition of semantic clones, rather a generic or more specific definition, which leads to target a certain subset of clones. This results in either poor precision or recall. We summarize the primary challenges in detection of semantic clone as follows:

- Lack of a well-established definition of semantic clones, and lack of a clear taxonomy of semantic clone types that set up the boundary between semantic clones, other clone types and false clones.
- There is no tool that can detect semantic clones with high precision and recall.
- There is no real semantic clone benchmark based on a well-defined definition that is used to evaluate semantic clone detectors.

In the previous chapter (Chapter 4), we used the .Net CIL (Common Intermediate Language) to detect clones across programming languages. The intermediate language, which is a lower level representation of the source code, holds all the semantics of the source code and could be useful to detect semantic clones since the definition of cross-language clones is closely related to semantic clones. In this study, we also use the .Net intermediate language to detect same language clones. We applied similar filters, extract the intermediate code features and represent them in ontology, finally we map the ontologies to detect clones.

We conducted several case studies on four datasets and we compared reported clones to other state of the art detectors by manually validating 3K clones. We observed that our approach is able to detect all types of clones at a comparable precision. Furthermore, we define a semantic clone taxonomy in which it defines all possible types of semantic clones. Then we used this taxonomy to create a set of 60 clones that we used to measure the recall of our approach and compare it to other clone detection tools. The results show that the existing tools cannot detect semantic clones accurately, while our approach is able to detect semantic clones with good recall. The contributions of this work are three folds:

- A novel tool (NetSim) to detect semantic clone at method level granularity with comparable precision and recall.
- A semantic clone taxonomy that defines different fine-grained types of semantic clones that we use to create semantic clones and evaluate semantic clone detectors.
- A comprehensive comparison of NetSim with the state-of-the-art tools.

This chapter is based upon the manuscript "Toward Semantic Code Clone Detection and Evaluation: The Capability of Microsoft .NET Intermediate language in Detecting Semantic Clones". It has been submitted to The Journal of Systems and Software. I am the lead author of the work under the supervision of Chanchal K. Roy. The manuscript has been modified and re-formatted to better fit the context of the thesis.

The remainder of the chapter is organized as follows. In Section 5.2 we provide the background about Semantic clones, Ontology and matching algorithms. Section 4.4 describes our proposed technique in clone detection. In Section 5.4, we evaluate our technique quantitatively and qualitatively. Section 5.5 discusses the related works and we summarize the chapter in Section 5.6.

5.2 Background

Similar to our previous work in chapter 4, we are using the .NET CIL to detect clones. However, in this chapter we are targeting same language clones and semantic clones. In this study, we represent the code in a simple Ontology (tree structure). We used LCS and Levenshtein algorithems for matching, that are already described in the previous chapter. Also, we using another matching algorithm in this study, Jaccard similarity. In this section, we present the definition of semantic clones, definition of Ontology and Jaccard similarity.

5.2.1 Semantic Clone Definition

A code fragment in the source code that identical or similar to another code fragment in the code base is considered code clone to the second and both called clone pair. This definition is based on the concept of similarity. In the literature, the following categorization of clone definition has been widely acceptable [140]:

- Type-1: Identical code fragments except for variations in whitespace (maybe also variations in layout) and comments.
- Type-2: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout, and comments.
- Type-3: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout, and comments.

- Type-4: Two or more code fragments that perform the same computation but are implemented through different syntactic variants.

Type I, Type II, and Type III clones are based on textual similarity. Code fragments are considered clones if they are textually similar even though they are functionally different. Textual clones are more common in a software code base because they are usually the result of copy/paste practice. Semantic clones are harder to detect since they could be implemented by different syntactic and any single change in a code fragment could change the meaning (functionality) of the fragment.

5.2.2 Ontology

Ontology is a conceptual model to represent knowledge of an application domain by first defining the relevant concepts of the domain and then using these concepts to specify properties of objects and individuals occurring in the domain [18]. In this work, we used disassembled code to build a knowledge model of the code base. We extracted entities, facts, relations, and other components from disassembled code and built a simple formal ontology. Methods are the main entities of the constructed ontology since we are targeting clones at the method level.

Ontologies could be defined in different ontology languages; the most popular one is OWL [37]. Defined ontologies usually consist of the following entities:

- *Classes or Concepts* are the main entities of an ontology. For example, it could be book, course or student. In code base domain, it could be file, class, method or variable.
- *Individuals* are instances of classes in the domain. All files in the source code are individuals.
- *Relations* identify the relationships between individuals.
- *Data types* specify values such as string and integer.
- *Data values* are simple values such as file name and method name.
- *Specialisation* represents inclusion relationship between two classes.
- *Exclusion* represents emptiness of intersection between classes.
- *Instantiation* represents membership between classes and individuals or values and data types.

5.2.3 Ontology Matching

Ontology matching is the process of finding the relationships or correspondences between entities of two different ontologies. In Figure 5.1 these correspondences are represented in dashed arrows. Sometimes practitioners refer to it as ontology alignment or mapping. Ontologies are represented in a hierarchy structure, XML-like, and are used basically to describe knowledge in semantic web, web services, or knowledge in other

domains. Ontology mapping plays a major role in different applications such as information sharing, query answering, data integration and so on.

Ontology matching techniques are based on finding the correspondences between ontologies' entities, structures, or relations. There is much written in this area. We implement our matching technique using similarity measurements, Levenshtein, LCS, and Jaccard to map elements between ontologies as detailed below.

5.2.4 Matching Algorithms

In this section, we demonstrate string matching algorithms used in our alignment technique. Many techniques could be used to compare strings depending on the way information is represented in the strings. For example, set of letters, sequence of letters, erroneous sequence of letters, set of words or sequence of words. In this paper, we used three different string similarity measurements. Levenshtein distance is used for sequence of letter strings, longest common subsequence is used for a sequence of words, and Jaccard similarity is used for a set of words. Other similarity measures could be used for the purpose of matching as Hamming distance, n-gram similarity, Euclidean, Cosine, Jaro–Winkler, Monge–Elkan, TFIDF, and Soundex. *Levenshtein Distance (levDist)* and *The Longest Common Subsequence (LCS)* are presented in Chapter 4.

Jaccard similarity (Jacc) or called gloss overlap between two strings is defined the intersection of their character set divided by the union of their character set [73] (Eq. 3). We used Jaccard coefficient to measure the similarity (overlap) between two sets of tokens.

$$Jacc(s_1, s_2) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|} \tag{3}$$

5.3 Proposed Technique

Our process starts with extracting code knowledge from the intermediate language of the source code and representing it in the descriptive knowledge language, ontology. The basic entity of the ontology is the method. Therefore, the semantic, syntactic and relationships of each method in the intermediate code are represented in our ontology. Then ontology matching is used to align similar ontologies. The matched ontologies represent the candidate clones for our proposed technique that we are evaluating by comparing it head-to-head with a number of state-of-the-art clone detectors. Figure 5.1 shows the overall processing steps for our proposed technique. A detailed discussion of all steps as follows.

(i) Intermediate Code Generation.

Our technique starts with disassembling the portable executable (PE) of the source code to generate the CIL. Therefore, the PE is a prerequisite in our technique and the source code needs to be compiled to generate the PE files. All artifacts that are used in clone detection are extracted from the intermediate code.

However, the source code is used in reporting and visualizing clones. Therefore, our technique is usable to detect clones from the executable files even when the source code is not available. In this case, we could identify the clones in the intermediate language and the source file name associated with the clone. But, we cannot identify a clone's start line, end line, or view the clone's source code. After CIL code is generated, we parse it to extract all method/function bodies (e.g. Fig 5.1-Column #2).

Table 5.1: Dataset used and the effect of filtering in reducing the CIL size

Data set	Domain	# of source files	# disassembled source files	# of extracted methods	CIL size	CIL extracted elements size
ASXGui	Video encoder GUI	76	14	160	2008 kb	200 kb
Netgore	Role-playing game	2038	1343	9267	56700 kb	5619 kb
OpenRA	Strategy game	673	617	2972	26200 kb	2578 kb
ScriptSc	Utility	314	136	512	10700 kb	356 kb

(ii) Ontology Population

CIL contains a large amount of information that needs to analyze in the comparison process of clone detection; CIL is more than ten times larger than its corresponding source code (Table 5.1). Therefore, the process of ontology population is based on extracting the necessary artifacts from the intermediate language that encompasses both syntactic and semantic information, which are relevant to clone detection. In this research we are using the CIL artifacts to populate CIL ontology. However, the source code itself contains a large number of artifacts that could be useful in clone detection and semantic clone detection as well.

In this step we parse the CIL and extract the concepts and relations that form the ontology. Figure 5.1, fifth column represents the schema of proposed CIL ontology. The method represents the base class (concept) of the ontology. For each method we create an ontology instance that consists of the following relations: Method name, method type, method arguments, method instruction set, and method call set.

The importance of this step comes in two folds: First, the reduction of data size used in the detection process. Table 5.1, shows the size of the extracted elements of CIL in KB. On average, data reduction is up to 30 times. Second, extracting elements that capture the syntax and semantic of the method. All of these elements encompass information about syntax or semantic of methods. Furthermore, All of them participate in the detection process and generate the final similarity measurement between methods.

(iii) Clone detection The general process of clone detection involves measuring the similarity between code fragments' (methods') extracted features to find matching methods (clones). Since we represent methods' metadata in ontology, clone detection is done through the process of ontology mapping. Different matching approaches could be used, as shown in Section 5.2.3. We design an ontology matching algorithm that aligns ontologies' elements which represent the methods semantic and structural.

Ontology has a hierarchical structure (Figure 5.1-Column #5). Every method is represented in an ontology

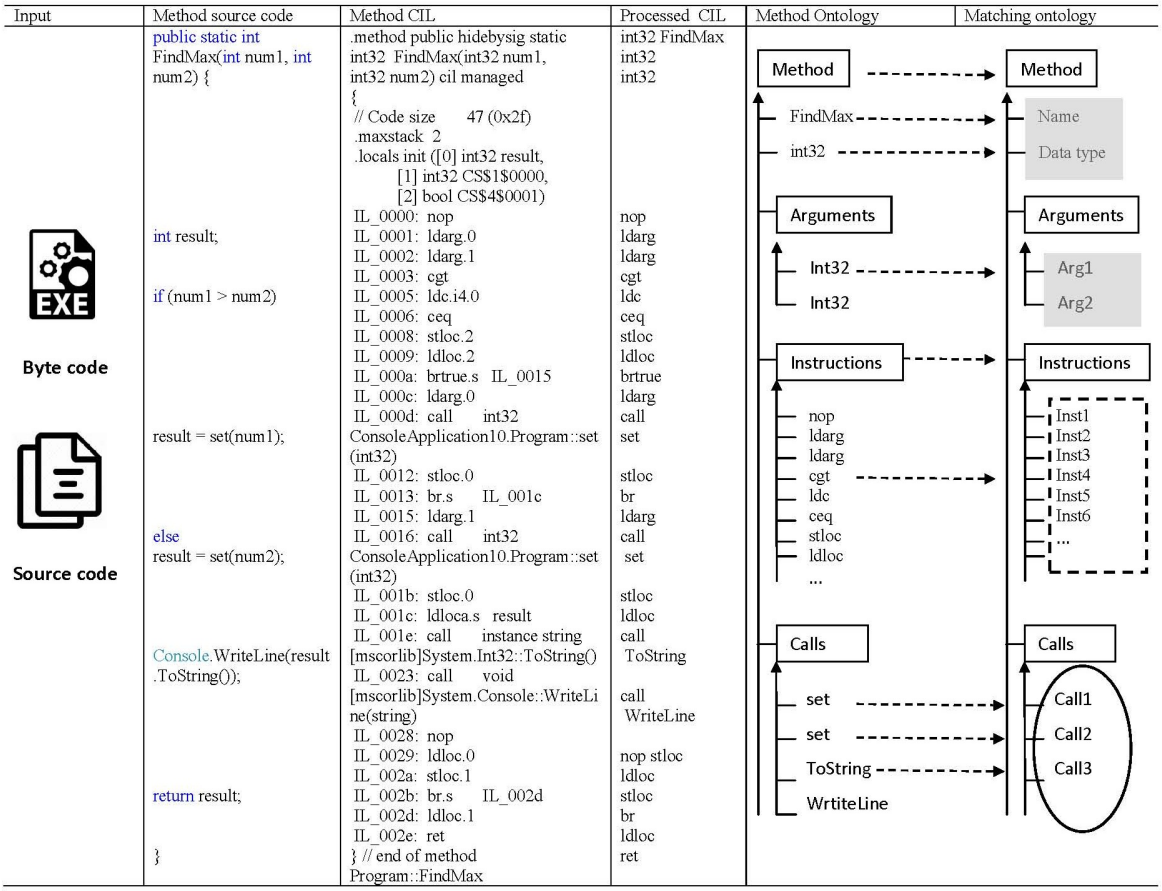


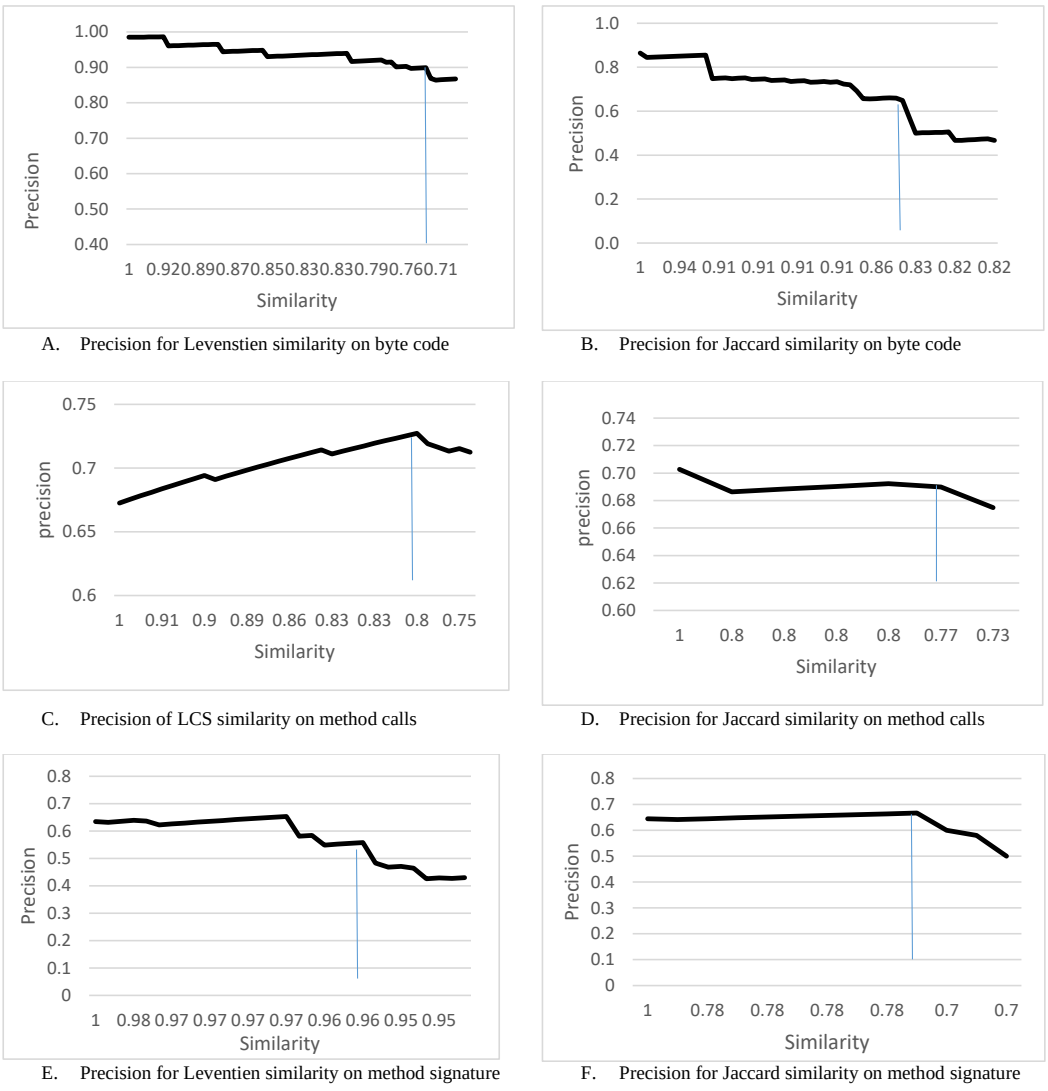
Figure 5.1: Simplified example of extracting ontology.

instance which all have a similar hierarchy structure except missing branch that is correlated to a missing artifact in the source code of the method. For example, a method that has no calls to other methods or has no arguments.

Our matching approach is based on aligning each branch of one ontology instance to its corresponding branch of a second ontology instance. This alignment is done by measuring the similarity between branches separately and all branches similarities participate in the final similarity value between two ontology instances. Different similarity measurements are used for different branches (artifacts) based on the type of data and relations. A similarity measurement algorithm has been chosen for each branch that enabled capturing and comparing the syntactic and semantic of those artifacts. Then, tune the weight of each artifact participation in the final similarity.

(vi) Similarity algorithm selection

Our proposed ontology contains the extracted byte code artifacts. We represent these artifacts in three main branches in the ontology (Figure 5.1). Then we match ontologies by measuring similarities between their corresponding branches. The first branch in the ontology represents method signature in the byte code (highlighted in Figure 5.1, Column 5: Method Ontology). The second branch represents byte code



A. Precision for Levenstien similarity on byte code

B. Precision for Jaccard similarity on byte code

C. Precision of LCS similarity on method calls

D. Precision for Jaccard similarity on method calls

E. Precision for Leventien similarity on method signature

F. Precision for Jaccard similarity on method signature

Figure 5.2: Precision measurement for different similarity algorithms.

instructions for the method (In a dashed rectangle in Figure 5.1, Column 5: Method Ontology). These instructions are filtered and processed as in [11]. Finally, the last branch represents methods called by this method (In Oval in Figure5.1, Column 5: Method Ontology). Then, all similarity measurement participate in the final decision to identify potential clones.

We used Levenshtein distance and Jaccard coefficient to measure the similarity between byte code instructions. Levenshtein distance is used at the granularity of a letter that enables capturing the syntactic and semantic between individual instruction since similar instructions have similar names with prefixes or suffixes. We used Jaccard coefficient as a set operator at words granularity to capture the commonality of instruction set between two methods.

For the other ontology components, called methods, we used the longest common subsequence (LCS) and

Jaccard similarity both at word granularity. Finally, for the signature branch, we used both Levenshtein and Jaccard similarity at words granularity.

(v) Threshold identification

In order to identify the optimal threshold for each similarity, we did an empirical analysis of these similarities on one system, ASXGUI (Table 5.1). In this experiment, we detect clones using one Ontology branch and one similarity measurement at a time. Then we evaluate the resulted clones manually to identify precision P (Eq. 4) over similarity. Precision is defined as the fraction of relevant instances, true positive (TP), among the retrieved instances, true positive plus false positive (TP+FP).

$$Precision(Ps) = \frac{TPs}{TPs + FPs} \tag{4}$$

As threshold value decreases, more clones are detected, and more false positives are introduced. Fig 5.2 shows the values of precision for different similarity measurements. Threshold is chosen when more clones have been detected at the highest precision. Therefore, the values for similarity thresholds are (0.75, 0.84, 0.80, 0.77, 0.96, and 0.78) respectively.

As part of our manual clone verification, we noted that other similarity measurements could be used to measure the similarity between byte code artifacts (ontology branches). Each similarity measurement has a different threshold but gives similar results; Look at the trends in charts A and B, C and D, and E and F.
(vi) Similarities combining

Multiple similarity measurements are used on three ontology components. The final decision to identify a method pair as a clone should consider all the similarities. Three scenarios are considered: 1) The final similarity value could be calculated as the summation of weighted similarities. 2) Method pair are considered clones if one of its similarities is above the corresponding threshold and 3) a method pair are considered clone if all of its similarities are above the corresponding threshold. After evaluating the clones of the three alternatives manually we found that the second alternative detects more true clones with a precision of 69%. Table 5.2 shows the number of clones reported and the manual evaluation results.

Table 5.2: Combination methods

Combination method	Weight of similarities	One similarity satisfied	All similarities satisfied
Total number of clone pairs	222	217	57
True positive	142	149	57

(vii) Report Generation

Our tool Generates XML files and HTML pages that show the reported clones with their source code.

5.4 Evaluation

To evaluate NetSim, we analyzed the results both from quantitative and qualitative perspectives. In a quantitative evaluation, we compare the number of clone pairs detected using NetSim to other detectors then we investigate common, extra and missed clones. In Qualitative evaluation, we measure the accuracy (precision) of reported clones and comparing it to other tools. Then we evaluate the ability (Recall) of NetSim in detecting semantic clones and compare it to other semantic and syntactic tools. Four open source systems are used in the evaluation process (Table 5.1), that are compiled and the CIL are generated.

5.4.1 Quantitative Evaluation

In this section, we conducted a head-to-head comparison of state of the art clone detectors as shown in Table 5.3. We selected clone detectors that detect any of the .Net languages (NiCad, Simcad, Simian, ConQat, and visual studio) for the sake of comparison. The goal of this experiment is to explore the ability of NetSim to detect clones that are detected by other clone detectors, and examine the ability to detect extra clones and missing clones.

Table 5.3: Clone detection tools and their configuration

Tool	Clone granularity	Output	Clone type	Configuration
NetSim	Method	Pair	1,2,3,4	Method granularity,minimum fragment size 5 lines, default similarity
NiCad	Block, method	Pair, class	1,2,3	Method granularity,minimum fragment size 5 lines, min 70% clone similarity, blind identifier normalization.
Simcad	block, method	Pair, class	1,2,3	Method granularity, greedy transformation, min 5 lines.
ConQat	Free (line level)	Class	1,2,3	Minimum fragment size 5 lines.
Simian	Free (line level)	Class	1,2	Default configurations.
Visual Studio	Free (line level)	Class	1,2,3	Default configurations.
MECC	Method	Class	1,2,3,4	Min similarity is 70% (Default id 80%) and MinEntry is 40 (Default is 50)
CCCD	Method	Pair	1,2,3,4	Default configurations.
Oreo	Method	Pair	1,2,3,4	Default configurations.

NetSim detects clones at method's level and reports clones in clone pairs. The comparison process with other tools that detect clones at the same granularity and report clone in pairs is straight forward. Figure 5.3 shows the results of comparing resulted clone pairs to both Nicad and SimCad in the four target systems.

It shows the number of common clone pairs detected by both NetSim and the other detectors, number of unique (extra) clones detected by NetSim, and number of missed clone by NetSim.

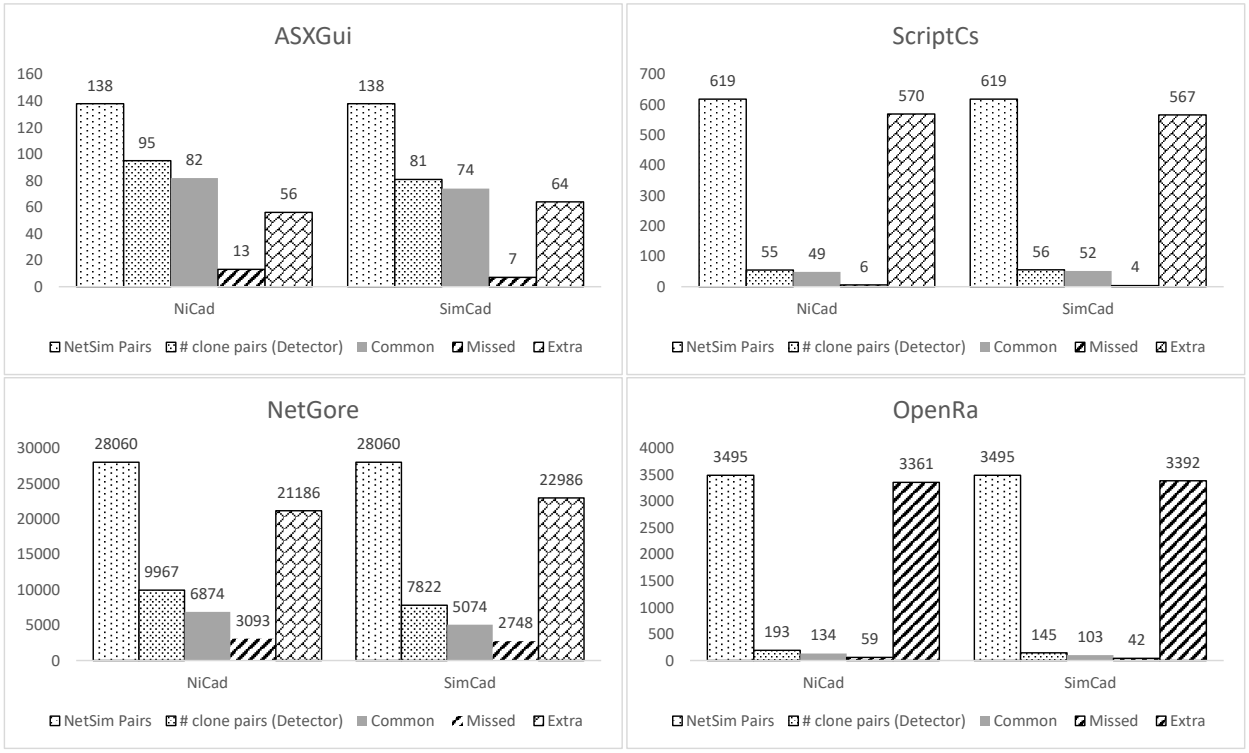


Figure 5.3: Comparing the number of clone pairs detected in our tool to NiCad and SimCad in four open source systems

To compare our clone results to some of the selected tools (Simian, ConQat, and Visual Studio) we faced two challenges. First, the output of these tools are clone classes, not clone pairs. It is not feasible to compare clone pairs to clone classes because not all combinations in the cluster result in clone pairs. The combinations of all clones in the cluster of clones Type-1, Type-2 and Type-4 results in clone pairs, but not clones Type-3. Let's say a cluster has three clone fragments A, B and C. A is clone Type-3 of B and B is a clone Type-3 of C. It is not necessary that A is a clone of C. The other challenge is these tools detect clones at a different granularity, line-level (free granularity).

Bellon et al. [24] used the *ok* and *good* metric to measure the overlapping between two clone pairs of different granularities. Since we could not generate an accurate clone pairs report for these tools we measured that common cloned line of code (LOC). Figure 5.4 shows the total clones LOC detected by NetSim, total cloned LOC by the selected tools and the common cloned LOC between NetSim and the selected tools. For most parts, our tool detects more clones and cloned LOC. Also, it detects most clones that are detected by other tools. We note from the figure that ConQat has more cloned LOC than other tools and that is due to the fact that ConQat does not filter out comments from clone fragments.

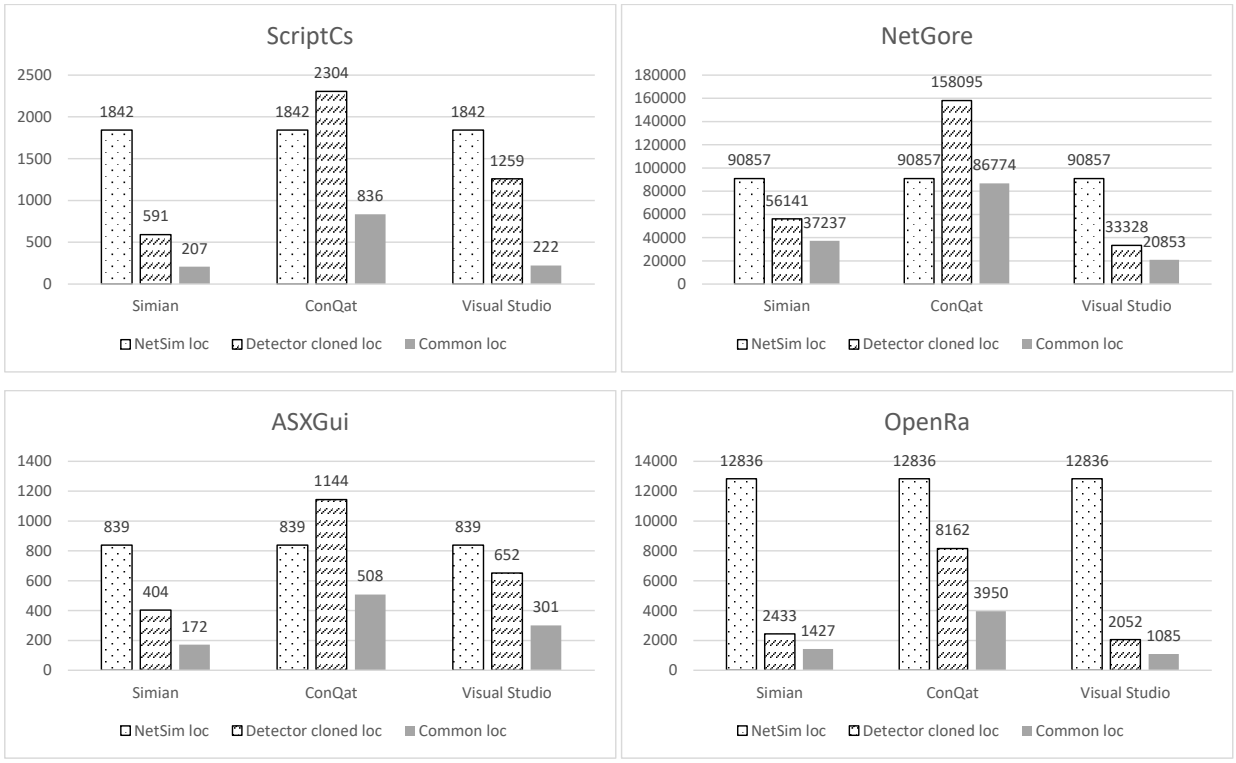


Figure 5.4: Comparing the number of cloned line of code in our tool to other clone detectors (Simian, ConQat, and Visual Studio) in four open source systems

5.4.2 Qualitative Evaluation

To measure the quality of reported clones we used both recall and precision for NetSim and compare it to other state-of-the-art.

5.4.2.1 Precision

Precision measures the accuracy of the results. It represents the percent of true clones in the reported clones. We measured the precision of the clone detection tools by manually validating a sample that is selected randomly for each tool.

Sample collection: We created a clone pool that contains all clones reported by the selected tools. We filtered out the duplicated clones that are reported by two or more tools to avoid double validation. For each tool, we randomly selected 500 clone pairs for validation, with a total of 3000 clone pairs.

Manual evaluation: Two C# developers were tasked with validating these clones as true positive or false positive. The judges were kept unaware of the tool that was used and used their own judgement for determining true positives and true negatives.

Precision: For each tool, the manual evaluation identifies how many true positives and false positives in the sample collected for that tool. Table 5.4 shows that the precision of NetSim is comparable to the

other tools. On the other side, NetSim targets a different type of clones (semantic clones). Besides, results in Section 5.4.1 indicated that NetSim reported more clones than other tools, which means NetSim able to detect more true clones that are not detected by the other tools.

Table 5.4: Comparing the precision based on sample evaluation

Tool	NetSim	NiCad	SimCad	ConQat	Simian	VS
Precision	0.72	0.84	0.78	0.54	0.76	0.59

5.4.2.2 Semantic Recall Using Mutation-Injection

Manual evaluation (Sections 5.4.1 and 5.4.2.1) indicates that NetSim is able to detect more clones than other tools with comparable precision. To measure recall of NetSim and compare it to other tools we faced The following challenges:

1. There is no benchmark for semantic code clones.
2. There is no benchmark for the .Net languages (C#, VB, F#, C++).
3. Our tool needs to generate the byte code for clones in order to detect clones i.e. the benchmark's system must be executable.

To overcome these limitations, we manually created a semantic clone benchmark then we inject these clones into a software system to be able to compile the system and generate the byte code. Following is detail for this experiment:

A. Semantic clones creation:

Similar to Roy et al. [140, 139], mutation operators are defined to create code clones, semantic clones for this experiment. We started by reviewing semantic clones in the literature[53, 150, 130], semantic clone samples in the available benchmarks[149, 153, 156] and semantic clones reported by tools [92, 87, 173, 103, 148, 149]. Then we defined a semantic clone taxonomy as follows:

Taxonomy of Semantic Clones

We show examples of semantic clones at the granularity of method that computes the average length of words in a sentence. Figure 5.5(a) shows a possible original method and the other examples are possible semantic clones. The purpose of these examples is to show different types of semantic clones and what type of mutation is used to create this type of clone. In these examples, we attempt to provide simple examples as much as possible for illustration purpose. Therefore, some of the semantic clones may still be a syntactic clone. However, applying extensive changes using the proposed mutators produces semantic clones.

- **Reordered clones**: This type of clone is created by reordering statements, declarations or both without affecting the functionality of the source code, see Figure 5.5(b). By applying more reordering will eliminate the syntactic similarity and preserve the semantic similarity.

- **Insertion clones**: Insertion of unnecessary code, intermediate results or both without affection the functionality of the source code, see Figure 5.5(c).

- **Deletion clones**: Deletion of unnecessary code or intermediate results create semantic clones, see Figure 5.5(d).

- **Expression clones**: Expression clone is resulted from changing any arithmetic or logical expression into an equivalent one, see Figure 5.5(e).

- **Nested/chain method call clones**: This type of clones appear when there is a hierarchy of method calls in the source code. These calls could be represented in a different hierarchy/chain with an equivalent semantic, see Figure 5.5(f).

- **If/switch replacement clones**: Representing if-else-if statement into switch statement and vice versa is a semantic clone, see Figure 5.5(g).

- **Type widening clones**: Type Widening or type narrowing to variable in a method create a semantic clone as shown in Figure 5.5(h).

- **Recursive clones**: Recursive clones result from converting loops into recursive structure or vice versa. This type of clone is difficult to detect using most of syntactic and semantic tools because the clone has different syntax and different execution path, see Figure 5.5(i).

- **Inline clones (Relative code)**: We called this type inline clone because of the ability to extract a segment of the code into a method, see Figure 5.5(j).

- **Control clone**: In control clones one control statement is replace by an equivalent control statement, see Figure 5.5(k).

- **Construct clones**: Any functionality in programming language could be solved using more than one construct. By construct we mean the problem representation (data structure) and algorithm selected. This results in syntactically a totally different clone. This type of clone is difficult to detect by most proposed techniques. Figure 5.5(l) shows an example of a cloned method that uses a pattern matcher to provide the same functionality.

- **Combine/collapse nested if clones**: Nested if(-else-if) statements could be restructured in a different but equivalent hierarchy that results in a syntactically different but semantically equivalent clone. Figure 5.5(m) shows an example.

- **Return value clones**: Figure 5.5(n) represents a cloned method with the same functionality except it prints instead of returns the result. Input/output-based techniques [97, 47, 130] fail to detect such clones.

- **Multi-mutant clones**: Semantic clones could result by applying one or more of the aforementioned operators (mutant operations). Combining two or more of these mutant operations makes the clones harder to detect. Figure 5.5(o) provides an example.

a- Original clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum+=w[i].length();
    avg=(double)sum/w.length;
  } else {
      avg=0.0;
  }
  return avg;
}
```

b- Reordered clone

```
double avgLength(String s) {
  s=s.trim();
  int sum=0;
  double avg;
  String[] w= s.split("\\s+");
  for(int i=0; i<w.length;i++)
       sum+=w[i].length();
  if(s.length()>0) {
    avg=(double)sum/w.length;
  } else {
      avg=0.0;
  }
  return avg;
}
```

c- Insertion clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  int intermediate=1;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
       { int l = w[i].length();
       intermediate**;
       String str= w[i];
       sum+=l;
       System.out.println(str);}
    avg=(double)sum/w.length;
  } else {  avg=0.0;  }
  return avg;
}
```

d- Deletion clone

```
ouble avgLength(String s) {
 double avg;
 int sum=0;
 . . .
 if(s.length()>0) {
   String[] w= s.split("\\s+");
   for(int i=0; i<w.length;i++)
        sum+=w[i].length();
   avg=(double)sum/w.length;
 }
 . . .
 return avg;
  }
```

e- Expression clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>=1) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum=sum+w[i].length();
    avg=(double)sum/w.length;
  } else {
      avg=0.0;
  }
  return avg;
}
```

f- Nest/chain call clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  if(s.trim().length()>0) {
  String[] w=s.split("\\s+");
  for(int i=0; i<w.length;i++)
       sum+=w[i].length();
  avg=(double)sum/w.length;
  } else {
     avg=0.0;
  }
  return avg;
}
```

g- If/switch replacement clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  switch (str.length()) {
  case 0: avg=0.0;
  default:
   String[] w=s.split("\\s+");
   for(int i=0; i<w.length;i++){
        sum+=w[i].length();
   }
   avg=(double)sum/w.length;
  }
  return avg;
 }
```

h- Type widening clone

```
float avgLength (String s) {
  float avg;
  float sum=0.0f;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum+=w[i].length();
    avg=sum/w.length;
  } else {
      avg=0.0f;
  }
  return avg;
}
```

i- Recursive clone

```
double avgLength(String s,int n){
  double avg;
  double sum=0;
  str=s.trim();
  if(s.indexOf(" ")>0) {
    String word=s.substring(0,s.indexOf(" "));
    String newStr=s.substring (s.indexOf(" ")+1);
     sum=word.length()+(n-1)*
          avgLength(newStr,n-1);
   }else {
      return (double) s.length();
   }
    avg=sum/n;
    return avg;
  }
```

Figure 5.5: Examples of semantic code clones.

j- Inline clone

```
double avgLength(String s) {
double avg;
s=s.trim();
if(s.length()>0) {
 String[] w= s.split("\\s+");
 avg=(double)sum(w)/w.length;
  } else {
       avg=0.0; }
  return avg;
}
static int sum(String[] s) {
  int sum=0;
  for(int i=0;i<s.length;i++){
      sum+=str[i].length();
    }
 return sum;
  }
```

k- Control clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    int i=0;
    while(i<words.length) {
       sum+=words[i].length();
    }
    avg=(double)sum/w.length;
  } else {
      avg=0.0;
  }
  return avg;
}
```

l- Construct clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
     Pattern p=
        Pattern.compile("\\w+");
     Matcher m=p.matcher(str);
     int counter=0;
     while(m.find()) {
        sum+=m.group().length();
        counter++;   }
     avg=(double)sum/counter;
   } else {
       avg=0.0;
  }
    return avg;
}
```

m- Combine/collapse nested if clone

```
double avgLength(String s) {
  double avg=0.0;
  int sum=0;
  str=str.trim();
  if(str.length()>0) {
    if (str.length()>1) {
     String[] w=s.split("\\s+");
    for(int i=0; i<w.length;i++)
        sum+=w[i].length();
    avg=(double)sum/w.length;
       } else {
          avg=0.0;
        }
   }
   return avg;
}
```

n- Return value clone

```
void avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum+=w[i].length();
    avg=(double)sum/w.length;
  } else {
      avg=0.0;  }
  System.out.println(avg);
}
```

o- Multi-mutant clone

```
double avgLength (String s) {
  double avg;
  int sum=0;
  int count=0;
  str=str.trim();
  String[] w= str.split(" ");
    for(int i=0;i<w.length;i++){
      if (w[i].length()>0) {
        count++;
        sum+=w[i].length();
        }
     }
  avg=(double)sum/count;
  return avg;
 }
```

Figure 5.5: Examples of semantic code clones (cont).

Figure 5.5 shows an example of each of the clone types. These examples are constructed to show a simplified example of clone types. In reality, most clones are resulted by applying more than one mutation as shown in Figure 5.5(o).

The 14 types of semantic clones defined in Section 5.4.2.2 are used to create 60 C# clone pairs. The clones are created manually by a graduate student who is not involved in the research. These 60 clones are recreated in C and Java for the purpose of evaluating detectors that do not support C#. The clones are available at. [1] Most of clones are created by more than one mutation operator.

B. Injection: The 60 semantic C# clones are injected randomly 10 times for a total of 600 unique reference clones. Each clone pair is injected at once then we ran the selected detection tool before we evaluate the results of the tool. We iterated this process for the 600 clone pairs. The number of times the tool is able to detect the injected clones divided by 600 represent the tool's recall.

[1] https://drive.google.com/open?id=1sK9XWQ3-fLTwVa2fWaBfV1BheHwaNFK8

C. Results: Table 5.5 shows the measured recall for the selected tools. We selected three state-of-the-art syntactic clone detectors (NiCad, SimCad, and Semian) and three state-of-the-art semantic clones (CCCD, MeCC, and Oreo). NetSim overcomes all the selected tools with a recall 85%. The syntactic tools have a low recall, NiCad was able to detect 20%, SimCad detects 16% and Simian detects 8% of the injected semantic clones.

We also measured the recall for three semantic clone detection tools, CCCD [100], MeCC [90], and Oreo [142]. Both CCCD and MeCC are semantic clone detection tools for C language and Oreo targets Java semantic clones. Therefore, the same clones are converted into both C and Java languages. We used the same procedure of injecting each clone 10 times in a total of 600 unique injections. Results in Table 5.5 shows that CCCD can detect 58% of semantic clones, MeCC can detect only 10% of semantic clones, and Oreo can detect 44.5% of semantic clones. The results again show the superiority of our proposed approach, NetSim, while having comparable precision, it has the highest recall among all the tools.

Table 5.5: Comparing the recall based on semantic clone injection

Tool	NetSim	NiCad	SimCad	Simian	CCCD	MeCC	Oreo
Recall	0.855	0.205	0.165	0.083	0.588	0.10	0.445

5.5 Related Work

Studies that targeted semantic clones tried to capture the semantic knowledge in the source code, represent it, and use it to detect clones. A number of these studies [93, 92, 98, 109, 105, 171] captured the data flow information and control flow information in the program that is represented in PDG. Then similar subgraphs (isomorphic subgraphs) were identified to be the candidate semantic clones. Other studies [116, 94] applied metric-based approaches to extract metrics from PDG and used these metrics to identify semantic clones.

PDG-based approaches are beneficial for non-contiguous code and insertion and deletion code. In addition, they could detect certain types of near-miss clones. Conversely, these approaches are extremely expensive in terms of PDG generation and subgraphs isomorphic. However, recent studies tried to reduce subgraph isomorphic costs in PDG [131, 53, 38, 66, 65, 148, 35]. For example, Qui et al. [131] built the PDG from the binary code and reduced the comparison cost of subgraph isomorphic by reducing the size PDG through filtering out unmatched subgraphs. Similarly, Gabel at el. [53] overcome the costs of subgraph isomorphic by converting it into tree similarity. While Henderson and Podgurski [38] used greedy independent subgraph measures to reduce subgraphs comparison costs. Sheneamer and Kalita [148] extracted source code features from PDG and AST and representing them in vectors where they use a machine learning classifier to identify similar vectors. Finally, Crussell et al. [35] represent the features of a PGD in a vector then used Locality Sensitive Hashing (LSH) to determine similar group vectors. They used this technique to detect similar Android applications.

Other studies have tried to analyze the behaviour of the source code. Some of them simply identified methods that generate the same output from the same input as clones [97, 47, 130]. Others [102, 81, 80] conducted more analysis on the behaviour of the source code by analyzing the execution path then identified the methods that have the same or similar execution paths as clones. Krutz and Shihab [102] used *diff* as a similarity measure to compare traces while Kamiya [81] used a frequent item-set algorithm for comparison. More analysis and transformation to the source code were performed by Mateev et al. [114]. They applied symbolic analysis in which the source code is transformed repeatedly until it becomes tractable (one execution path from input to the output) then they compared the normalized code to detect clones. This approach is applicable for simple programs that only could be transformed into tractable form.

In a recent study by Saini et al. [142] they used machine learning to detect semantic clones. In their model they used the clones reported by SourcererCC [144] as training set for their model. Their approach is good for detecting a good proportion of Type-3 clones. But, it is still not accurate in detecting semantic clones. Besides, it needs a high hardware configurations (Min 12G of memory), and works for Java files in the second level of directory structure only. Ours is a novel approach with intermediate language and ontology mapping. Furthermore, we proposed a taxonomy of semantic clones and conducted extensive evaluation with the state-of-the-art.

5.6 Summary

In this work, we developed NetSim, a semantic clone detection tool for the .NET programming languages. NetSim is an ontology-based knowledge representation for byte code. It extracts the basic Knowledge that represents the syntax and semantics from the byte code and represents it in a lightweight ontology. We compared NetSim against six competing clone detection tools by measuring and comparing their precision and recall. The experiment shows that NetSim is able to detect significantly more semantic clones in the code base with a comparative precision to other detectors. These extra clones that are only detectable by NetSim make its recall the highest among all detectors. Furthermore, we defined comprehensive edit scenarios (a semantic clone taxonomy) that were used to create a broad range of possible types of semantic clones. We used the defined taxonomy to create a set of 60 semantic clones that are used to evaluate NetSim and compare it to other tools.

The biggest challenges we faced in evaluating our technique are 1) the unavailability of semantic clone benchmarks, and 2) no benchmark for the .NET programming languages. Due to the need of having a semantic clone benchmark that enables the evaluation of existing and emerging techniques and tools, we propose a methodology to build a benchmark with minimal human effort for validation. Then we create a semantic benchmark for four programming languages, as presented in the next chapter (Chapter 6).

Chapter 6

SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge

Both newly proposed code clone detection techniques and existing techniques and tools need to be evaluated and compared. This evaluation process could be done by assessing the reported clones manually or by using benchmarks. The main limitations of available benchmarks include: they are restricted to one programming language; they have a limited number of clone pairs that are confined within the selected system(s); they require manual validation; they do not support all types of code clones. To overcome these limitations, we proposed a methodology to generate a wide range of semantic clone benchmark(s) for different programming languages with minimal human validation. Our technique is based on the knowledge provided by developers who participate in the crowd-sourced information website, Stack Overflow. We applied automatic filtering, selection and validation to the source code in Stack Overflow answers to create a dataset of semantically similar code fragments from the alternative answers to the same programming problem. Finally, we build a semantic code clone benchmark of 4000 clones pairs for the languages Java, C, C# and Python.

6.1 Introduction

There are two primary measurements used to evaluate the accuracy of clone detection tools, precision, and recall. First, validate tool results as true positive and false positive manually. This evaluation can measure the actual precision (results accuracy) of the tool. However, it is time-consuming and dependent on the individual's understanding of code clone definition. Besides, it does not reflect all the true clones exist in the target system (Recall). Measuring Recall of tools is more challenging since there is a need to know all true clones in the target system, i.e., clone benchmark. Building a clone benchmark is a challenging task since it needs to be accurate, large enough, includes all types of clones and represents real clones that occur during the development process.

Existing benchmarks have the following issues: First, they are not large enough or not enough references for all types of clones (Both syntax and semantic); for example, Bellon's benchmark [24] does not have type-4 clones, Krutz's benchmark [101] has only 66 clones without any type-1 clones, Yuki's benchmark [176] has 19 unclassified clones and Wanger's benchmark [167] has only 29 type-4 clones. Second, they are the resulted clones of one or more detection tools [24]. That means only a subset of real clones in the systems that are

detected by selected tools. Third, they follow the definition of the creator [139], which do not represent real clones generated by developers. Fourth, existing benchmarks supports one or two languages only. For example, BigCloneBench [153] supports Java only and Krutz's benchmark [101] is for C language. Still there is no benchmark specialized for semantic clones.

The two primary challenges in building a clone benchmark are candidate selection and manual validation. Candidates should be real clones that are occurred in the development process. Also, selection should be objective and includes all types of clones, i.e, not author's defined clones or some tool's results. Manual validation is a nontrivial process and time consuming [168]. For example, Jeffery et al. [153] spent 600 working hours to validate their benchmark.

In this chapter, we propose a methodology for building a code clone oracle (functional clone database) with the minimal need for manual validation. We used the knowledge in Stack Overflow, a question and answer website for computer programming [169]. We extracted all questions that have a code snippet as well as the answers that have code snippets. Code snippets within the answer for one particular question are considered code clones because they answer the same problem and perform the same functionality. Programmers who have tried or tested these code snippets are able to distinguish the good responses from the bad responses by using the voting mechanism provided by Stack Overflow. We consider the voting process as our manual validation since it reflects programmers' opinions regarding the quality and functionality of the code snippet. Also, we perform more filtering and processing on the selected code such as minimum size limit, syntax correctness, and syntactical clone filtering. Then, we evaluated more than 10000 clone pairs manually. The results show that more than 53% of the selected method pairs are real semantic clones. Finally, we were able to build a semantic clone benchmark, SemanticCloneBench, that contains 4000 semantic code clones for the languages Java, C, C# and Python.

This chapter is based upon the manuscript "SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge" [12]. It was published by myself, Chanchal K.Roy and Tonghao Chen at the 14th International Workshop on Software Clones (IWSC 2020). I was the lead author of the work under the supervision of Chanchal K. Roy, while Tonghao Chen contributed to the manual validation. The publication has been modified and re-formatted to better fit in the context of the thesis.

The remaining of the chapter is organized as follows. Section 6.2 presents our methodology of building semantic clone benchmark. Section 6.3 describes the benchmark and it's usage. In section 6.3.2 we analyse the textual similarly of the clones in the benchmark. Section 6.5 discusses related works and section 6.4 shows threats to validity, followed by summary in section 6.6.

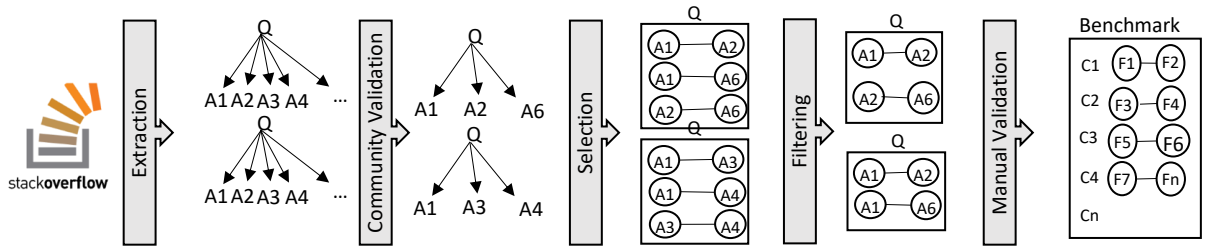


Figure 6.1: A schematic diagram that shows the complete process for constructing a semantic code clone benchmark using data from Stack Overflow

6.2 Methodology

Stack Overflow is a website that enables users to ask and answer questions related to programming languages. It has over 19 million questions, 28 million answers, 73 million comments, and 11 million registered users[1]. For each question, participants submit different answers. We can be sure that correct answers should be functionally equivalent. On the other hand, some questions do not have multiple answers, answers could be incomplete or erroneous, answers could not represent a fully functional unit, or answers could be syntactically similar which does not comply with the definition of the semantic clone. Therefore, we design a selection process that includes automatic validation, filtering, and selection to overcome all the aforementioned challenges and reduce the number of candidate clones that need manual validation. The complete process construction of our semantic code clone benchmark is summarized in Figure 6.1.

6.2.1 Data Extraction

Our process starts by collecting the Stack Overflow questions and answers from ***SOTorrent*** [21] database. A Stack Overflow thread contains a question, several answers, a number of comments, votes for answers, and other related data. We were interested in posts (Questions) that represent a programming problem and have multiple answers. Our query retrieved all answers that solve programming language problem (Contain source code) and grouped by question. Listing 6.1 shows our query to retrieve all C answers. We collected answers for four programming languages, Java, C, C#, and Python. These answers create the initial clone class.

Our query retrieved approximately 4 million answers from ***SOTorrent***. Table 6.1 column# 2 and 3 show the total number of questions and answers in Stack Overflow. The results of our query are shown in column# 4 and 5.

[1] `https://data.stackexchange.com/stackoverflow/queries`

Listing 6.1: An example of SQL Query

```
SELECT init.Body as Answer
      ,init.ID as Post_Link
      ,parent.ID as Id
      ,parent.Title as Title
FROM [sotorrent-org:2018_12_09.Posts] init
      LEFT JOIN [sotorrent-org:2018_12_09.Posts]
      parent on init.Parentid = parent.Id
Where init.Body like '%<code>%'
      AND init.PostTypeID = 2
      AND parent.Tags like '%<c>%'
```

Table 6.1: Number of questions and answers in Stack Overflow and each processing step

Filter	No Filter (Total Number)		>=2 answers contain source code		Include methods		>=10 lines		Votes>0	
Language	Questions	Answers	Questions	Answers	Questions	Answers	Questions	Answers	Questions	Answers
Java	1556189	2582119	432086	1183439	95837	435186	30153	92788	15320	60374
C	297405	602044	119990	338519	26597	90202	10435	30702	6972	21236
C#	1310182	2216152	375898	1025420	52827	176384	12180	33211	8615	26777
Python	1223511	1807837	359051	968235	57820	219632	23935	70485	15227	49886

6.2.2 Syntax Validation

Answers in Stack Overflow contain both text and code snippet. Fortunately, the code snippet in each answer on Stack Overflow is surrounded by a $< code >$ tag. This greatly reduces the difficulty of code extraction, as it only needs to apply a regular expression to match the content between $< code >$ and $< /code >$ and store them separately in the code file.

Code snippets in answers come in different granularities. Some are just a line of code or a few lines that show how an API is used. Others come as a function or a full program. Most of code snippets are not a complete program. Therefore, a solid functional unit should be extracted that is error-free.

For the purpose of building a semantic code clone benchmark we have chosen method/function level granularity, that is because the non-function code snippets are usually too short to express the meaning behind it. On the contrary, the method usually contains more information and the process of the solution. Therefore, the method will be more helpful in determining clones. TXL [34] is used in this step to extract methods from the code snippets. TXL parser extracts syntactically correct methods only. Also, we used TXL to normalize the selected methods. This includes the removal of comments, white spaces and a strict pretty printing. Table 6.1 column# 7 shows the number of answers that contain methods. In some cases, some answers contain more than one method so we saved all methods in the answers for further selection process.

We also performed a minimum-line-limit filter in this step to make sure all extracted functions have at least 10 lines of code [156, 132]. This also helped us to filter out about 75% of the candidate functions, and will reduce the manpower for manual validation later. Refer to Table 6.1 column #9.

6.2.3 Functionality Validation

After all the aforementioned selection processes, we still have a large number of methods to validate (92K Java methods, 30K C methods, 10K C# methods, and 21K Python methods). In order to reduce manual validation, we applied another filter as per the quality of the answers by the Stack Overflow community.

Stack Overflow users participate in asking questions, answering, commenting, and voting in order to get rewards. Stack Overflow employs a voting mechanism for questions and answers to help users to identify trustworthy answers [55]. Answers are posts that have up-votes and down-votes with the best answers receiving more up-votes. For the purpose of our benchmark we considered only up-voted methods and ignored non-voted and the down-voted methods. Up-voted methods represent 60% of total methods so this step reduces the total number of methods by 40%, see Table 6.1 (last two columns).

6.2.4 Syntactic Clone Filtering

The main goal of this study is to build a real semantic clone benchmark. Therefore, we reviewed all the definitions of semantic clones in the lecture and we found that most definitions agree that semantic clones are code fragments that perform the same computation but are implemented through different syntactic variants [141]. Therefore, we identified syntactically similar fragments using NiCad [138] and eliminate them. NiCad has high precision and recall in detecting syntactic clones [156]. It is true that the eliminated answers are functionally equivalent. But, they are not considered semantic clones according to the definition of semantic clones.

Table 6.2: Number of clone pairs after syntactic filtering

Language	Clone pairs (biggest method)	Detected By NiCad	**Candidate clones**	Clone pairs (same name)	Detected By NiCad	**Candidate clones**
Java	18146	3141	**15049**	7,627	2,019	**5,944**
C	6210	312	**5898**	3,383	372	**3,011**
C#	7404	488	**6916**	1,986	523	**1,507**
Python	15103	2773	**12343**	7,322	2,451	**5,320**

Before using NiCad to detect syntactic clones we build all possible clone pairs. For each question, all answers combinations are candidate clones. However, some answers have more than one method. We used two heuristics to select between methods. First, we chose the biggest method in the answer. Table 6.2, the second column shows the number of candidate clone pairs. Second, we selected methods in answers that have the same name. Column #5 shows the number of candidate clones that have the same name.

Table 6.2 shows the number of syntactic clone pairs detected by NiCad. Clones that are detected by NiCad represent syntactic clones (around 18%) are filtered out for the purpose of the semantic clone benchmark.

```
static int convertToInt(string a)
  {
    int x = 0;
    Char[] charArray = a.ToCharArray();
    int j = charArray.Length;

    for (int i = 0; i < charArray.Length; i++)
    {
      j--;
      int s = (int)Math.Pow(10, j);

      x += ((int)Char.GetNumericValue(charArray[i]) * s);
    }
    return x;
  }
```

```
static int convertToInt(string a)
{
  int x=0;
  for (int i = 0; i < a.Length; i++)
    {
      int temp=a[i] - '0';
      if (temp!=0)
      {
        x += temp * (int)Math.Pow(10, (a.Length - (i+1)));
      }
    }
  return x ;
}
```

Figure 6.2: Semantic clone pair form Stack Overflow post "How can I convert String to Int? " [2]

It is evident that there is a good number of syntactic clones exist in Stack Overflow answers. These clones are perfect to construct a syntactic benchmark. To avoid the bias of using clone detector, we can select a random set of answers and validate them manually or we can select answers that are textually similar and validate them manually.

6.2.5 Manual Validation

Table 6.3: Number of validated clone pairs

Language	Number of validated clones	True semantic clones	Validation time(hours)
Java	1775	1000	26
C	1742	1000	32
C#	1894	1000	26
Python	2020	1000	30
Total	7431	4000	114

Manual validation is the last task for building benchmarks. We hired two judges to mark all method pairs as true positive or false positive according to their functionality. A graphical interface is designed to facilitate the validation process. It displays the candidate clone, the title of Stack Overflow question to give an idea of the validation functionality, and three options, *True*, *False*, and *Undecided*. The judges were summer students who were not involved in the research. Rather, they were introduced to the topic of semantic clones and read a number of related papers. Judges are asked to tag candidate clones as *True* if they perform the same functionality and *False* otherwise. Our final goal was to have 1000 clone pairs for each programming language. Table 6.3 shows the number of candidate clones that are needed to be validated to reach 1000 semantic clone pairs for each language. Also, it shows the time needed to finish validation. A total of 114

[2]https://stackoverflow.com/questions/1019793/how-can-i-convert-string-to-int

hours spent by two judges to validate 7432 clone pairs.

Manual validation proves the feasibility of our selection process in finding semantic clone candidates. 54% of selected candidates are true semantic clone. Unlike Krutz and Le[101], they examine 1536 candidate clones to identify 66 clone pairs, only 9 semantic clones for one programming language. Also, the selection and filtering process helped to reduce the effort in building a benchmark of 4000 clone pairs. Only 114 hours are needed by two judges to finish the validation load. Finally, Figure 6.2 shows an example of a semantic clone that is tagged as true by the judges.

6.3 SemanticCloneBench in use

We designed our semantic code clone benchmark (SemanticCloneBench) into two forms, injected in a system form and stand-alone clones form. SemanticCloneBench is available online for use [3].

Injected in a system. We designed SemanticCloneBench as a real software system that has clones. We selected 4 systems (Table 6.4) and we injected the clones into random locations. We considered medium size systems since we know not all detection tools are scalable for large systems. We injected clones into syntax-correct locations, where no clone pair are injected in the same file. Each subject system is associated with the clone references.

Table 6.4: Subject system to inject clones

System	Language	Number of files	Line of code
JHotDraw7	Java	711	130k
PostgreSQL-12.0	C	1343	1368k
Mono1.1.4	C#	9822	5518k
django	Python	2031	240k

Stand alone clones. We also kept each clone pair in a single text file for another usage. Practitioners could use a subset of clones for other testing or could inject them into other systems. Some clone detectors cannot scale for large systems and others have certain limitations. For example, Oreo [142] detect clones only in the second directory of the file structure.

6.3.1 Evaluating Clone Detectors using SemanticCloneBench

SemanticCloneBench can be used to measure the semantic recall of clone detection tools. Recall is defined as the fraction of the true clone detected by a clone detection tool (1), where $Recall_{Sem}$ is the measured semantic recall, D_{clone} is the set of detected clones by the tool and R_{clone} is the set of reference clones in SemanticCloneBench.

[3] https://drive.google.com/open?id=1KicfslV02p6GDPPBjZHNlmiXk-9IoGWl

$$Recall_{Sem} = \frac{D_{clone} \cap R_{Clone}}{R_{Clone}} \qquad (1)$$

We used SematicCloneBench to measure semantic recall for four clone detectors (See Table 6.5). We used NiCad to filter out syntactic clones in an earlier stage of building the benchmark. We are using it again with different configurations. We increased the dissimilarity threshold (UPI) of NiCad from 0.3 to 0.35 and used blind renaming of identifiers to enable NiCad to detect more gaped clones. We ran NetSim on the injected Mono1.1.4 with the 1000 clones. But, NetSim detection is based on the byte code and the source code of Mono1.1.4 could not compile even when clone injection is syntactically correct. Therefore, we injected one clone at a time, compile the code base then run NetSim. We found out that there are 100 clones are compileable. Out of 100 injected and compiled clones, NetSim was able to detect 48 clones. We ran NiCad and SimCad on the injected JhotDraw7 with the 1000 clones. NiCad was able to detect 40 clones while SimCad detected 25 clones. Oreo runs on source code in the second level of the file structure only. Therefore, we had to inject the clones in certain locations for Oreo. Oreo is able to detect only 97 clones.

We have not expected a good recall for clone detection tools on SemanticCloneBench since most tools are based on measuring syntactic similarity to identify code clones. SemanticCloneBench represents the region where most detection tools are difficult to perform. That is because it follows the definition of semantic clones.

Table 6.5: Clone detection tools' recall

Tool	Granularity	Target language	Clone type	Recall
NetSim	Method	C#	1,2,3	0.48
NiCad	Method	Java, C, C#, and Python	1,2,3	0.04
SimCad	Method	Java, C, C#, and Python	1,2,3	0.025
Oreo	Method	Java	1,2,3 and 4	0.097

6.3.2 Textual Similarity in SemanticCloneBench Clones

Semantic clones represent a challenge for most detection tools where it is hard for most tools to reach a good recall since semantic clones are not textually similar. Most tools that are based on the similarity threshold, set up the threshold value at the point where false positive started to produce. These tools never attempt to detect clones with a lower threshold in order to keep high precision. However, semantic clones carry some textual similarity. But it is still not examined by detection tools. In this section, we measured the textual similarity of clones in SementicCloneBench.

We start by normalizing the code in the benchmark. This includes removal of comments and white-spaces, normalizing all literals and identifiers and pretty-printing of the code. Then we used the Longest Common Subsequence (LCS) [71] algorithm on the sequence of normalized lines to measure the textual similarity

between clones. We scale the similarity in the range of 0 to 100, where 100 means textual identical and 0 means total-textual different. Figure 6.3 shows the distribution of clones in SemanticCloneBench over the similarity scale.

The figure shows that semantic code clones hold a low textual similarity. The majority of Java, C, and C# semantic clones are less than 50 textual similar. This means that clone detection tools should set their threshold to a value less than the range in order to detect clones in that range. The figure shows that Python semantic clones have more textual similarity. Most Python clones reside in the range of 30 to 60. But it is still very low comparing to the threshold used by detection tools which justify the shortage of most detection tools to detect semantic clones.

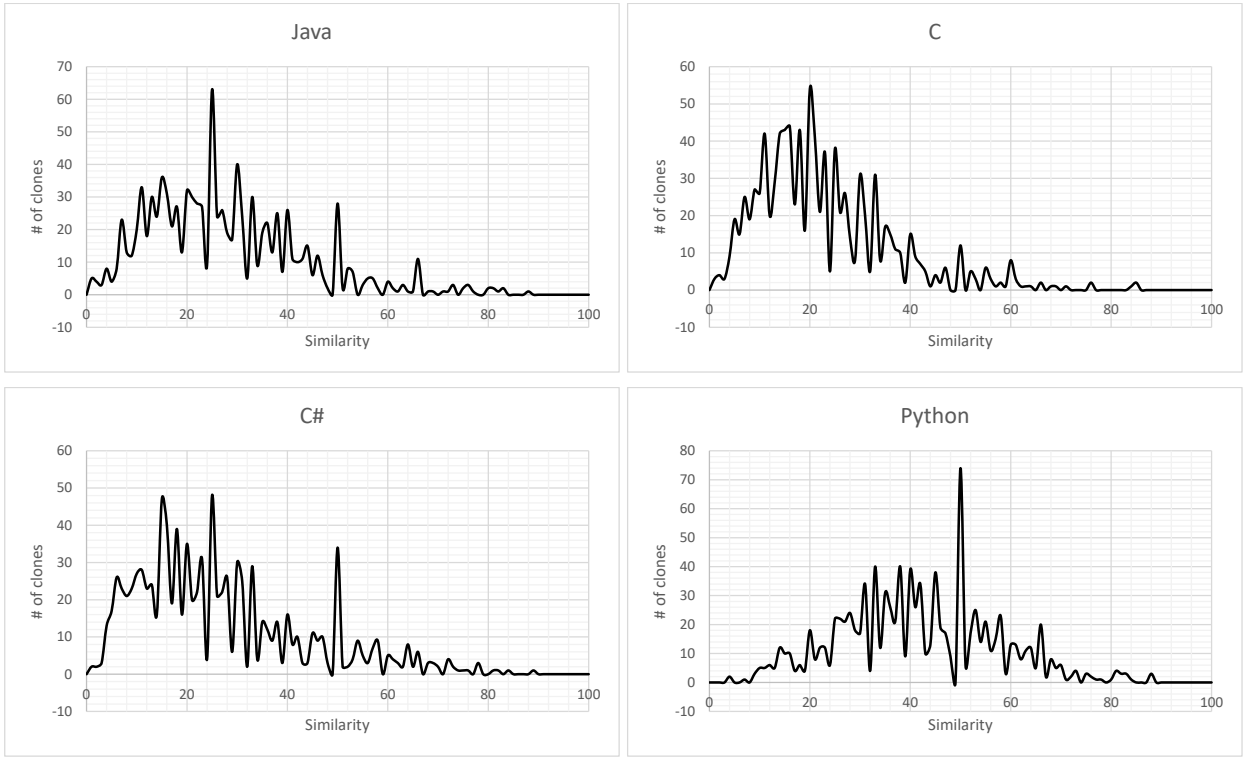


Figure 6.3: Textual similarity between the clones of SemanticCloneBench

6.4 Threats to Validity

The clones in our benchmark may not be real clones. However, these clones are provided as example solutions for many problems by many real developers. Furthermore, we considered the method level clones and those received high votes by the stack Overflow users. Thus, the clones in our benchmark might be real semantic clones to some extent. In order to guarantee further accuracy, judges also validated the clones. Of course, there might be errors in the manual validation process and in future we plan to involve more judges for

cross validation. In filtering out syntactic clones (type-1, type-2 and type-3 clones) we used the NiCad clone detector. NiCad might not have filtered out all such clones. However, NiCad has been widely used for detecting such types of clones. Furthermore, even if there are some syntactic clones in the semantic benchmark, they may not impact much in evaluating the semantic clone detection tools. Of course, the presence of more syntactic clones may wrongly promote some tools for their capability of detecting semantic clones. We plan to address this concern with a future release of the benchmark.

6.5 Related Work

The first clone benchmark was created by Bellon et al. [24] in 2007. They verified 2% of clones reported by six clone detection tools in four C systems and four Java systems. This verification was performed by one judge. The main issue in this benchmark is created by one author and all candidates are clone reports by tools. Therefore, not all types of clones are included in this benchmark.

Similarly, Krutz and Le [101], selected a random set of 1536 method pairs in three C open source programs, *Apache*, *Python* and *PostgreSQL*. They hired three expert judges and four students to evaluate these pairs manually. They found that only 66 clone pairs, out of the 1536 candidate clones, were real clones. Their benchmark contains 43 type-2 clones, 14 type-3 clones and 9 type-4 clones (semantic clones).

Svajlenko at al. [153] hired 9 judges to manually tag candidate clones as true positive or false positive. They identified 44 common functionalities that are used widely in *IJaDataset*. Unlike the aforementioned benchmarks, they used a search heuristic to identify candidate clones, for these functionalities, instead of using clone detection tools. They were able to build a large benchmark that contains all types of clones. However, their benchmark is only for Java language.

Roy and Cordy [139] design various scenarios to create different types of clones that are injected into the code base and used in the evaluation process to measure recall and precision. However, these scenarios have to be comprehensive (covering all types of clones that could appear in real source code) and not depend on any clone definition.

Recently, Yuki et al. [176] proposed a technique to build a benchmark through mining software versions and identify merged methods (merged cloned methods in the next version). Two merged methods are considered clones if they are textually similar and called by the same methods in the next version. Unfortunately, their technique is limited to only refactored clones, which are a small portion of code clones. In more than 15K versions they were able to identify 19 clones only.

Other studies [167, 149, 50] used Google Code Jam [4] as a semantic clone benchmark. They considered solutions for the same question semantic clones. The correct solutions that pass judgments of the contest are truly semantic programs. But the majority of clone detectors do not work on a file or a program granularity. Therefore, benchmarks should be built on a more fine-grain granularity such as block or method. Wanger et

[4]https://codingcompetitions.withgoogle.com/codejam

al. [167] used the main functions of the solutions as semantic clones to be able to evaluate CCCD [100]. Then they build a semantic clone benchmark by considering the main method only and excluding syntactically similar methods. Their benchmark includes 29 clone pairs only (16 Java clones and 13 C clones).

Stack Overflow is well known as a source of online code clones where developers copy source code from projects into Stack Overflow and vice versa [8, 172]. A number of empirical studies used clone detection tools to identify code cloned from Stack Overflow and analyze its quality [51, 15, 132]. These studies found clear evidence that a reasonable portion of Stack Overflow reused code is buggy code, outdated code, has security flow, or violate software licenses.

6.6 Summary

Semantic code clones are the most challenging type of clones to detect. Also, evaluating semantic clone results is fuzzy according to the definition and understanding of semantic clones. In this paper, we mine the knowledge of the Stack Overflow to find semantic code clones with minimal human effort. We considered the answers for the same programming question are a functionally equivalent code and methods in the answers are semantically similar answers. We extracted methods from Stack Overflow answers. Then we filter out small, incomplete and low-voted answers by Stack Overflow community. We validated 7431 candidate pairs to build a semantic clone benchmark of 4000 clone pairs in the languages Java, C, C# and Python. These clone pairs could be used as a piece of knowledge to understand how semantic clones are syntactically different and would guide the research to enhance the detection of semantic clones.

Manual validation proofs the feasibility of our selection process in finding semantic clones candidates. 54% of selected candidates are true semantic clone. On the other hand, the selection and filtering process helped to reduce the effort in building a benchmark of 4000 clone pairs. Only 114 hours are needed by two judges.

Our technique benefits from the on-line community in several ways:

- *Select functionality*

  Programmers look for online help to solve certain programming functionality. Programmer's lack of knowledge in a certain area is the main reason for cloning. Therefore, Stack Overflow questions reflect real functionality where cloning happens.

- *Creating clones*

  Users of different programming skills and backgrounds compete to solve Stack Overflow questions. Therefore, there are more than one answer for each question, and these answers are real functional clones that are created by online community.

- *Identify duplicate questions*

Stack Overflow advises users to avoid duplicate question. In case of duplication, Stack Overflow links the duplicated questions.

- *Validate clone*

  Stack Overflow motivates users and awards them according to their participation in asking, answering, commenting, voting, tagging, and editing questions. More experienced users generally edit more questions, suggest better answers, and vote for the best answers. The voting process is considered an acceptable validation for question correctness and functionality.

Chapter 7

Evaluating Semantic Code Clone Detectors Using an Injection Framework

Software clones is an active research area. A large number of clone detection tools have been proposed and are still being proposed in this area. These tools target all types of clones. However, a smaller number of clone detection tools target semantic clones. Although some work has been done to evaluate and compare clone detection tools, little work has been done to evaluate semantic clone detection tools. This is mostly due to the difficulty in identifying semantic clones themselves and building a real semantic clone benchmark. In this study, we propose an injection framework that injects semantic clones in a software system, and evaluates the semantic clone detectors results.

7.1 Introduction

Code clones are classified either as syntactic clones (Type-1, Type-2, or Type-3 depending on their syntax similarity) or semantic clones (Type-4), which are syntactically different but still perform the same computation [141]. We discussed in Chapter 2 a number of semantic clone definitions and their vagueness. We showed that some definitions are narrow, such as Simions, while other definitions are broad, such as similar functional clones. Progressively, clone detectors are produced by practitioners due to the variety of their applications and their importance [155]. Studies that target semantic clones should be base on a clear definition of semantic clone. Existing clone detection tools and newly proposed tools need to be evaluated to prove their efficiency in detecting clones according to a real semantic clone benchmark.

There are two primary measurements used to evaluate the accuracy of detection tools, precision and recall. To measure the precision of a tool, the detected clones need to be validated as true positive or false positive manually. Manual validation is time-consuming and dependent on the individual's understanding of the code clone definition. Manually validating detected clones does not reflect the true clones in the target system (recall). Measuring recall of tools is more challenging since there is a need to know all true clones in the target system, i.e., clone benchmark.

Building a clone benchmark is a challenging task since it needs to be accurate, large enough, include all types of clones and represents real clones that occur during the development process. Existing clone benchmarks have a good number of syntactic clone references. But they have a very limited number of

semantic clones. This is due to the difficulty of finding real semantic clones either manually by practitioners or automatically by tools. In this study we define a comprehensive taxonomy for semantic clones, then we define mutants to create clones according to the taxonomy, then we inject these clones into a subject system. Finally, we use these clones to evaluate semantic clone detection tools. Furthermore, we used the semantic clone benchmark that is created in the previous section with our framework to evaluate the tools.

This chapter is based upon the unpublished manuscript "Evaluating semantic code clone detectors using Semantic Clone Benchmark". I am the lead author of the work under the supervision of Chanchal K. Roy. The manuscript has been modified and re-formatted to better fit in the context of the thesis.

The remainder of the chapter is organized as follows. In Section 7.2 we present our taxonomy of semantic clones which we use to create a semantic clone benchmark. Section 7.3 describes our injection and evaluation framework, then we use the framework on our benchmarks to evaluate tools in Section 7.4. We measure the precision and the execution time of the tools in Sections 7.4.1 and 7.4.2. Finally, Section 7.5 discusses related work and Section 7.6 summarizes our work.

7.2 Taxonomy of Semantic Clones

In Chapter 5, we defined the semantic clone taxonomy, and then we used the taxonomy to create a semantic clone benchmark of 60 clones in C, Java, and C#. In this section, we demonstrate the taxonomy and discuss in detail the process of creating the semantic clones.

According to Roy and Cordy [141], semantic clones are defined as code fragments that perform the same computation but are implemented through different syntactic variants. Unlike syntactic clone, that are resulted from copy/paste/modify, semantic clones are are created unintentionally where they do not look similar textually. In this section, we propose a comprehensive taxonomy of all possible types of semantic clones. Our taxonomy is not built based on a simple copy/paste/edit scenario, but rather, it is derived from a large body of published work on existing semantic clone definitions [53, 150, 130], semantic clone types [140, 60, 167] and reported semantic clones by tools and benchmarks [92, 87, 173, 103, 148, 149, 153, 156].

We show examples of semantic clones at granularity of method that computes the average length of words in a sentence. Figure 7.1(a) shows the original possible function and the others are all possible semantic clones. The purpose of these examples is to show the types of semantic clones and what type of mutation is used to create this type of clone. In these examples, we attempt to provide simpler examples as much as possible for the purpose of elaboration. Therefore, some of them still may be syntactic clones. However, applying extensive changes using the proposed mutators may produce semantic clones.

- **Reordered clones**: This type of clone is created by reordering statements, declarations or both without affecting the functionality of the source code, see Figure 7.1(b). By applying more reordering will eliminate the syntactic similarity and preserve the semantic similarity.

- **Insertion clones**: Insertion of unnecessary code, intermediate results or both without affecting the functionality of the source code, see Figure 7.1(c).

- **Deletion clones**: Deletion of unnecessary code or intermediate results creates a semantic clone, see Figure 7.1(d).

- **Expression clones**: Expression clone is resulted from changing any arithmetic or logical expression into a different but equivalent one, see Figure 7.1(e).

- **Nested/chain method call clones**: This type of clones appear when there is a hierarchy of method calls in the source code. These calls could be represented in a different hierarchy/chain with an equivalent semantic, see Figure 7.1(f).

- **If/switch replacement clones**: Representing if-else-if statement into switch statement and vice versa is a semantic clone, see Figure 7.1(g).

- **Type widening clones**: Type Widening or type narrowing to variable in a method create a semantic clone as shown in Figure 7.1(h).

- **Recursive clones**: Recursive clones result from converting loops into recursive structure or vice versa. This type of clone is difficult to detect using most syntactic and semantic tools because the clone has different syntax and different execution paths, see Figure 7.1(i).

- **Inline clones (Relative code)**: We called this type inline clone because of the ability to extract a segment of the code into a method, see Figure 7.1(j).

- **Control clone**: In control clones one control statement is replaced by an equivalent control statement, see Figure 7.1(k).

- **Construct clones**: Any functionality in programming language could be solved using more than one construct. By construct, we mean the problem representation (data structure) and algorithm selected. This results in a syntactically totally different clone. This type of clone is difficult to detect by most proposed techniques. Figure 7.1(l) shows an example of a cloned method that uses a pattern matcher to provide the same functionality.

- **Combine/collapse nested if clones**: Nested if(-else-if) statements could be restructured in a different but equivalent hierarchy that results in a syntactically different but semantically equivalent clone. Figure 7.1(m) shows an example.

- **Return value clones**: Figure 7.1(n) shows a cloned method with the same functionality except it prints instead of returns the result. IO-based techniques [97, 47, 130] fail to detect such clones.

- **Multi-mutant clones**: Semantic clones could result by applying one or more of the aforementioned operators (mutant operations). Combining two or more of these mutant operations makes the clones harder to detect. Figure 7.1(o) provides an example.

Figure 7.1 shows an example of each of the semantic clone types. These examples provide simplified examples of semantic clone types. However, in reality, most clones result from applying more than one of the mutations, as shown in Figure 7.1(o).

a- Original clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum+=w[i].length();
    avg=(double)sum/w.length;
  } else {
      avg=0.0;
  }
  return avg;
}
```

b- Reordered clone

```
double avgLength(String s) {
  s=s.trim();
  int sum=0;
  double avg;
  String[] w= s.split("\\s+");
  for(int i=0; i<w.length;i++)
       sum+=w[i].length();
  if(s.length()>0) {
    avg=(double)sum/w.length;
  } else {
      avg=0.0;
  }
  return avg;
}
```

c- Insertion clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  int intermediate=1;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
       { int l = w[i].length();
       intermediate**;
       String str= w[i];
       sum+=l;
       System.out.println(str);}
    avg=(double)sum/w.length;
  } else {  avg=0.0;  }
  return avg;
}
```

d- Deletion clone

```
ouble avgLength(String s) {
 double avg;
 int sum=0;
 . . .
 if(s.length()>0) {
   String[] w= s.split("\\s+");
   for(int i=0; i<w.length;i++)
        sum+=w[i].length();
   avg=(double)sum/w.length;
 }
 . . .
 return avg;
  }
```

e- Expression clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>=1) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum=sum+w[i].length();
    avg=(double)sum/w.length;
  } else {
      avg=0.0;
  }
  return avg;
}
```

f- Nest/chain call clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  if(s.trim().length()>0) {
  String[] w=s.split("\\s+");
  for(int i=0; i<w.length;i++)
       sum+=w[i].length();
  avg=(double)sum/w.length;
  } else {
     avg=0.0;
  }
  return avg;
}
```

g- If/switch replacement clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  switch (str.length()) {
  case 0: avg=0.0;
  default:
   String[] w=s.split("\\s+");
   for(int i=0; i<w.length;i++){
        sum+=w[i].length();
   }
   avg=(double)sum/w.length;
  }
  return avg;
 }
```

h- Type widening clone

```
float avgLength (String s) {
  float avg;
  float sum=0.0f;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum+=w[i].length();
    avg=sum/w.length;
  } else {
      avg=0.0f;
  }
  return avg;
}
```

i- Recursive clone

```
double avgLength(String s,int n){
  double avg;
  double sum=0;
  str=s.trim();
  if(s.indexOf(" ")>0) {
    String word=s.substring(0,s.indexOf(" "));
    String newStr=s.substring (s.indexOf(" ")+1);
     sum=word.length()+(n-1)*
          avgLength(newStr,n-1);
   }else {
      return (double) s.length();
   }
    avg=sum/n;
    return avg;
  }
```

Figure 7.1: Semantic code clone taxonomy.

j- Inline clone

```
double avgLength(String s) {
double avg;
s=s.trim();
if(s.length()>0) {
 String[] w= s.split("\\s+");
 avg=(double)sum(w)/w.length;
  } else {
       avg=0.0; }
  return avg;
}
static int sum(String[] s) {
  int sum=0;
  for(int i=0;i<s.length;i++){
      sum+=str[i].length();
    }
 return sum;
  }
```

k- Control clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    int i=0;
    while(i<words.length) {
       sum+=words[i].length();
    }
    avg=(double)sum/w.length;
  } else {
     avg=0.0;
  }
  return avg;
}
```

l- Construct clone

```
double avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
     Pattern p=
        Pattern.compile("\\w+");
     Matcher m=p.matcher(str);
     int counter=0;
     while(m.find()) {
        sum+=m.group().length();
        counter++;   }
     avg=(double)sum/counter;
   } else {
       avg=0.0;
  }
    return avg;
}
```

m- Combine/collapse nested if clone

```
double avgLength(String s) {
  double avg=0.0;
  int sum=0;
  str=str.trim();
  if(str.length()>0) {
    if (str.length()>1) {
     String[] w=s.split("\\s+");
    for(int i=0; i<w.length;i++)
        sum+=w[i].length();
    avg=(double)sum/w.length;
       } else {
          avg=0.0;
        }
   }
   return avg;
}
```

n- Return value clone

```
void avgLength(String s) {
  double avg;
  int sum=0;
  s=s.trim();
  if(s.length()>0) {
    String[] w= s.split("\\s+");
    for(int i=0; i<w.length;i++)
         sum+=w[i].length();
    avg=(double)sum/w.length;
  } else {
     avg=0.0;  }
  System.out.println(avg);
}
```

o- Multi-mutant clone

```
double avgLength (String s) {
  double avg;
  int sum=0;
  int count=0;
  str=str.trim();
  String[] w= str.split(" ");
    for(int i=0;i<w.length;i++){
      if (w[i].length()>0) {
        count++;
        sum+=w[i].length();
        }
     }
  avg=(double)sum/count;
  return avg;
 }
```

Figure 7.1: Semantic code clone taxonomy (cont).

7.2.1 Semantic Clone Creation According to Semantic Clone Taxonomy

In this section, we used the semantic clone taxonomy we defined in the previous subsection to create different types of semantic clones. The 14 types of semantic clones defined in Figure 7.1 are used to create a set of semantic clones. These clones are not created by simply applying one of the mutations in the taxonomy definition. For example, following the first type in the taxonomy (Reorder clones) to reorder two statements in the code fragment will not produce a semantic clone. To comply with the definition of semantic clone, the created clone should meet two conditions. First, the produced code fragment should be textually different and this could be done by applying the same mutant multiple times or applying more than one mutant. Second, the produced fragment should preserve the functionality.

To overcome the limitations of automatic creation of semantic clones according to the taxonomy, we created our set manually. We hired a graduate student who is not involved in the research. The student have a good background of clone types and clone detection. We introduced him to our taxonomy. We asked him to create a set of semantic code clones according to the taxonomy by using one or more mutants.

A set of 60 clones are created in C#. Then these clones are recreated in C and Java for the purpose of evaluating detectors that do not support C#. We make sure all mutants are used in our semantic clone set. Table 7.1 shows the number of times each mutant appears in the created clones. The clones are available at. [1] Figure 7.2 shows an example of a semantic clone pair that is created using the defined mutants.

Table 7.1: Mutation operators usage in creating semantic clones

Number	Clone Type (Refactoring operation)	Frequency
1	Reordered clones	6
2	Insertion clones	8
3	Deletion clones	8
4	Expression clones	4
5	Nested/chain method call clones	7
6	If/switch replacement clones	6
7	Type widening clones	4
8	Recursive clones	4
9	Inline clones (Relative code)	5
10	Control clones	15
11	Consrtuct clones	10
12	Combine/collapse nested if clones	6
13	Return value clones	5
14	Multi-mutant clones	8

[1] https://drive.google.com/open?id=1sK9XWQ3-fLTwVa2fWaBfV1BheHwaNFK8

```java
public static String Pascal(int level)
{
   System.out.print("Pascal Triangle Clone: \n");
   String result = "";
   int[][] arr = new int[50][50];
   for (int i = 0; i < level; i++)
   {
      int k = level;
      while (k > i)
      {
         result += " ";
         k--;
      }
      int j = 0;
      while (j < i)
      {
         if (j == 0 || i == j)
         {
            arr[i][j] = 1;
         }
         else
         {
         arr[i][j]= arr[i-1][j] + arr[i- 1][j-1];
         }
         result += arr[i][j] + " ";
         j++;
      }

      result += "\n";
   }
   return result;
}
```

```java
public static String PascalTriangle(int no_row)
{
   System.out.print("Pascal Triangle : \n");
   int c = 1, blk;
   String result = "";
   for (int i = 0; i < no_row; i++)
   {
      for (blk = 1; blk <= no_row - i; blk++)
      {
         result += " ";
      }
      for (int j = 0; j <= i; j++)
      {
         if (j == 0 || i == 0)
         {
            c = 1;
         }
      else
      {
         c = c * (i - j + 1) / j;
      }
      result += " " + c;
   }
   result += "\n";
   }
   return result;
}
```

Figure 7.2: Example of a semantic code clone.

7.3 Injection and Evaluation Framework

In this section, we describe our injection and evaluation framework. Figure 7.3 shows the overall process of evaluation. Our framework evaluate detection tool based on any available benchmark. The process consists of four stages, clone selection from a benchmark, clone injection, run detection tool and detection evaluation.

Clone selection. Our process starts by selected a clone pair to inject into a subject system. The user needs to select a subject system to inject clones into and should select a set of clones to inject (Benchmark).

Clone injection. The subject system is parsed to find all valid locations for injection. Two random syntactically correct locations in the subject system are selected. We kept track of injected clone and locations for latter evaluation.

Tool execution. After injecting the clone pair into the subject system, we run the clone detection tool on the system.

Results evaluation. In evaluation, we look up the tool's clone report to decide if it is able to detect the injected clone. The recall of the tool is calculated as the number of successfully detected clones by the tool divided by the total number of injected clones.

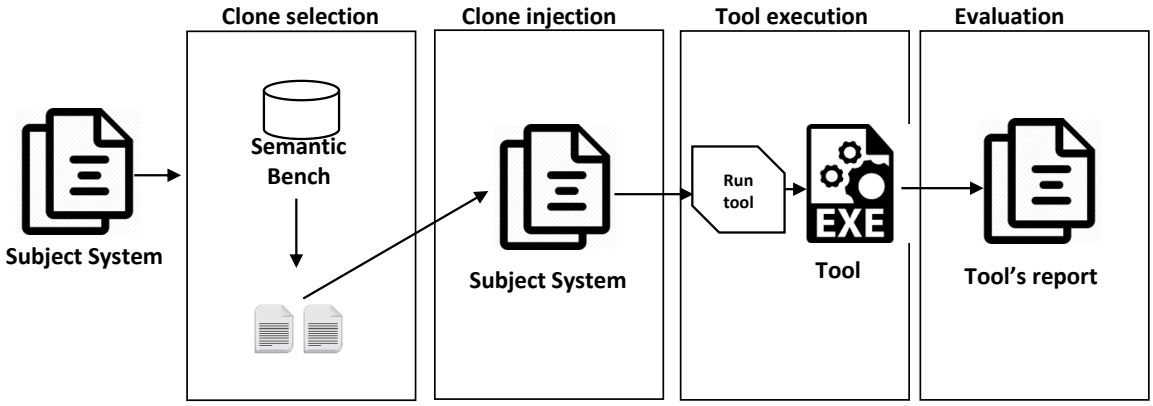


Figure 7.3: Evaluation framework

We make sure that the injection of each clone is done independently of other clones. That is done by injecting one clone at a time i.e. the process of injection is iterative over benchmark clones. After evaluating the injection of one clone, the subject system is reset to its original state for the next injection.

7.4 The Use of Framework

In this section, we used the framework to measure the recall for selected tools using two benchmarks, the taxonomy-based benchmark and the SemanticCloneBenchmark. We chose three semantic clone detectors, Oreo, Mecc and CCCD, and three syntax detectors, NiCad, SimCad and Simian. Table 7.2 summarises the selected tools plus our tool, NetSim.

Taxonomy-based evaluation In this experiment we used the 60 C# clones created according to the taxonomy in section 5.4.2.2. The 60 semantic C# clones are injected randomly 10 times for a total of 600 unique reference clones. Then we repeat the experiment by injecting each clone pair in 40 different random locations for a total of 2400 unique reference clones. Each clone pair is injected at once in a correct syntactic location. We make sure that each clone pair is injected into two different files. Then we ran the selected detection tool before we evaluate the results of the tool. We iterated this process for the 600 and 2400 clone pairs. The number of times the tool is able to detect the injected clones divided by 600 or 2400 represents the tool's recall. To avoid bias between tools we used the same injection locations for all tools.

The tools, NetSim, NiCad, Simian and SimCad support C#. But the tools CCCD and MeCC support C only and Oreo Support Java only. We performed the same procedure for these tools on the C and Java clones.

Results: Table 7.3 shows the measured recall for the selected tools. We selected three state of the art semantic clones (CCCD, MeCC and Oreo) and three state of the art syntactic clone detectors (NiCad, SimCad and Semian) . NetSim outperforms all the selected tools with a recall 85%. Results shows that CCCD can detect 58% of semantic clones, MeCC can detect only 10% of semantic clones, and Oreo can

Table 7.2: Clone detection tools

Tool	Granularity	Language	Output	Clone type	Configuration
NetSim	Method	C#, C++, VB, F#	Clone Pairs	1,2,3 and 4	Default configuration
MECC [90]	Method	C	Clone Classes	1,2,3 and 4	Min similarity is 70% (Default is 80%) and MinEntry is 40 (Default is 50)
CCCD [100]	Method	C	Clone Pairs	1,2,3 and 4	Default configuration, except min 5 lines
Oreo [142]	Method	Java	Clone Pairs	1,2,3 and 4	Default configuration
NiCad [138]	Block, Method	Java, C, C#, Python	Clone Pairs, Clone Classes	1,2,3	Method granularity, minimum fragment size 5 lines, min 70% clone similarity, blind identifier normalization.
Simcad [164]	Block, Method	Java, C, C#, Python	Clone Pairs, Clone Classes	1,2,3	Method granularity, greedy transformation, min 5 lines.
Simian [63]	Free (line level)	Java, C, C#, Ruby, C++, JS, Lisp, Cobol	Clone Classes	1,2	Default configurations.

detect 45% of semantic clones. The syntactic tools have a low recall, NiCad was able to detect 20%, SimCad detects 16% and Simian detects 10% of the injected semantic clones.

Table 7.3: Comparing the recall based on semantic clone injection

Tool	600-Injections	Detected	**Recall**	2400-Injections	Detected	**Recall**
NetSim	600	513	**0.855**	2400	2031	**0.846**
MeCC	600	60	**0.100**	2400	240	**0.100**
CCCD	600	353	**0.588**	2400	1422	**0.593**
Oreo	600	267	**0.445**	2400	1080	**0.450**
NiCad	600	123	**0.205**	2400	494	**0.206**
SimCad	600	99	**0.165**	2400	393	**0.164**
Simian	600	60	**0.100**	2400	243	**0.101**

SemanticCloneBench-based evaluation

In this experiment, we used both the injection framework and SemanticCloneBench to evaluate the selected tools. We performed a similar procedure to inject one clone at a time and evaluate the detection result of the tool. Our framework performs the injection, execution of the tool and evaluation results, for all clone pairs in the benchmark (1000 clones). Table 7.4 shows the recall of all the tools. The same process and injection is done for all tools. However, not all the selected tools support one programming language , as shown in Table 5.3. Therefore, we used the C# semantic bench for our tool (NetSim), NiCad, SimCad and Simian, C

semantic bench for MeCC and CCCD, and Java for Oreo.

Results show that our tool's recall is 0.48, which is the highest among all tools, followed by Oreo and CCCD. Mecc has the lowest recall among semantic clone detectors. As for syntax clone detectors, NiCad, SimCad and Simian, they were able to detect 18, 32 and 25 clones from the 1000 clones. Our tool works on the byte code to detect clones. Therefore, after each injection, the target system is built to generate the byte code. Only 100 clones out of the 1000 injected clones are able to compile and generate the byte code.

Table 7.4: Recall based on SemanticCloneBench and precision results

Tool	Language	Injections	Detected	Recall	Precision	F-score
NetSim	C#	100	48	0.48	0.72	0.58
MeCC	C	1000	25	0.025	0.85	0.05
CCCD	C	1000	59	0.059	0.50	0.11
Oreo	Java	1000	60	0.06	0.80	0.11
NiCad	C#	1000	18	0.018	0.84	0.04
SimCad	C#	1000	32	0.032	0.78	0.06
Simian	C#	1000	24	0.024	0.76	0.04

7.4.1 Precision for Semantic Clone Detectors

Measuring recall is not enough to have an unbiased comparison of clone detection tools. Some tools might have high recall at the same time it might have a lot of false positives i.e. large and inaccurate results. Therefore, we manually validate a sample of the tools' detected clones. We randomly selected 500 clone pairs for each tool and validated them as true or false positive. Table 7.4 shows the measured precision for the tools. The last column shows the f-score for each tool. F-score combines both recall and precision in one value, see Equation 7.1.

$$F\text{-}score = 2 * \frac{precision * recall}{precision + recall} \tag{7.1}$$

Results: The combination of recall and precision gives a good idea of the accuracy of the tools. The recall of all tools is low for the SemanticCloneBench. That is because SemanticCloneBench contains real semantic clones, that is they have no syntax similarity. In both benchmarks, NetSim achieves the highest recall. NetSim reported more clones than other tools with an accuracy of 70% and a recall of 85% and 48% on both benchmarks. Mecc reported fewer clones for the subject systems. Its clone report is 80% accurate but it has a very low recall for semantic clones. Manual evaluation shows that more than 95% of true validated clones by Mecc are syntax clones. CCCD reported more clones in subject systems; however, it has the lowest precision. On the other hand, it is good in detecting semantic clones. Manual validation showed that Oreo is good at detecting Type-2 and Type-3 clones. Syntax clone detectors, NiCad, SimCad and Simian are good

at detecting syntax clones.

All validated tools measure clones at the method level except Simian. Since both benchmarks' clones are at the method level, the framework evaluation stage is straight forward. Our framework considers the injected clone pair successfully detected by the tool if the tool's reported clone pair locations intersect with the injected clone pair locations. However Simian reports clones using a free granularity i.e. a pair of similar code snippets. Bellon et al. [24] used two measurements, *Ok* and *Good*, for the portion of the reported clone span (cover) for the injected clone. Simian detected 24 semantic clone pairs from SemanticCloneBench. In fact, it detected only one full semantic clone and the 23 reported clones are sub-clones i.e. just a portion of the clone pair.

7.4.2 Execution Time for Semantic Clone Detectors

Execution time is important in the selection of a clone detector based on the type of application it is used for. Semantic clone detectors need to process more code artifacts (AST, PDG, Byte code). Also, they perform more complex comparisons. In this section, we measure the execution time for the selected tools over three different systems' sizes, 10K, 100k and 1M lines of code (LOC). We executed all tools on the same machine, with 2.7GHz Intel Core i5, 8GB of RAM that runs Ubuntu 18.04.1. However, we used Windows 10 for NetSim as it uses the Visual Studio dis-assembler to generate the intermediate language of the source code.

Table 7.5 shows the execution time for both syntactic and semantic clone detection tools. Results show that syntax clone detection tools are faster than semantic clone detection tools. Simian is the fastest. But it detects Type-1 and Type-2 clones only. CCCD is the slowest tool among all tools and that is due to the type of processing it performs on the code (concolic analysis). Oreo uses machine learning, which requires a training stage before it starts processing the source code. Machine training with a big sample of clones takes more than 2 minutes, which explains the long execution time for the 10K system. On the other hand, Oreo scales well with larger systems of 100k and 1M LOC. But, we found Oreo does not process all files in a larger file structure. MeCC is very fast as semantic clone detector. But we notice it does not scale well with larger systems. Also, our manual validation shows that the majority of clones detected by MeCC are syntactic clones. NetSim has no problem scaling for larger systems. But, it still takes a long time as it works on the byte code and performs three comparisons.

Table 7.5: Execution time for tools

Tool/LOC	10K	100K	1M
NetSim	8s	13m31s	5h45m14s
MeCC	1s	2m33s	ERR
CCCD	17m24s	2d3h23m21s	ERR
Oreo	2m24s	2m34s	2m48s
NiCad	0.7s	6.4s	2m34s
SimCad	3s	20s	8m22s
Simian	0.3s	8s	2m14s

7.5 Related Work

Recently, many tools have been proposed to detect clones [53, 74, 14, 98, 151, 27]. The main contribution of these tools is to detect more clones (semantic clones) that are not detectable by most state-of-the-art tools [166, 162, 108, 175, 174, 115, 49, 104]. Some tools' validation does not include precision or recall measurements [104, 166]. Some of the tools are validated by measuring precision manually on a random sample of clones [53, 74, 14, 98]. Others measured recall based on non-semantic benchmarks [162, 107, 175, 25, 170].

Roy and Cordy [139] proposed a mutation and injection framework to evaluate clone detection tools by measuring both recall and precision. They defined an edit taxonomy to create all types of clones. Then clones are created automatically according to the taxonomy. We performed a similar injection and evaluation procedure, except we used SemanticCloneBench and evaluated semantic clone tools. Jeffrey and Roy [155] developed a mutation and injection framework similar to Roy and Cordy [139]. Similarly, clones are created automatically, injected, the tool is executed, and recall is measured. They used the framework to evaluate a tool's ability to detect syntactic clones. They also used BigCloneBench [157] and Bellon's [24] bench to evaluate the same tools and compare the results of different benchmarks. Our study differs because it uses a semantic clone benchmark alone and measures the semantic clone recall of the tools.

7.6 Summary

In this chapter, we presented the injection and evaluation framework. This framework takes a clone benchmark, a target system and a tool to evaluate as input. It automatically injects clones, executes the tool and evaluates the reported clone to measure the tool's recall. We used our framework to evaluate four semantic clone detection tools, NetSim, MeCC, CCCD and Oreo, plus three syntax clone detectors, NiCad, SimCad and simian. Results showed that NetSim able to detect more semantic clones followed by CCCD and Oreo. We used SemanticCloneBench created in Chapter 6 for the evaluation purpose. Also, we used a second benchmark that is created manually based on the semantic code clone taxonomy we defined. We also measure and

compare tools' precision by manually validating 500 clones for each tool. Results show that NetSim has the highest f-score among other tools. Manual validation shows what type of clones each tool is able to detect. Finally, we measure the execution time for the tools. Results show that semantic clone detection tools do not perform well with big systems either in detection or accuracy.

Chapter 8

Thesis Conclusion

Code clones represent a significant portion of a system's source code. Code clones are classified as syntactic clones or semantic clones. Syntactic clones have syntax similarity and usually result from copy-paste-edit operations while semantic clones have no syntax similarity but have same functionality. Code cloning has been an active research area over the last couple of decades due to its importance in software development, quality assurance, and mainly software maintenance. Despite a large number of studies in the area of code cloning (including clone detection, analysis, management, and benchmarking), fewer studies target semantic clones. Semantic clone detection has been one of the biggest challenges in the area. In our research, we mainly focused on the area of semantic clones, and more specifically semantic clone analysis, semantic clone detection, semantic clone benchmarking, and detection tools evaluation.

The remainder of this chapter is organized as follows. Section 8.1 summarizes the contents of this thesis. Section 8.2 presents the key contributions of the thesis. Section 8.3 outlines directions for future research made possible by this thesis. Finally, Section 8.4 provides a summary of the chapter.

8.1 Research Summary

In this thesis, we presented our research, which focused on semantic clone detection, benchmarking, and evaluation. We began with an empirical study of code cloning in open source games (Chapter 3). In this study, we investigated the cloning status in game systems, compared cloning in game versus non-game systems, and investigated cloning across games. In the second dimension of this study, we analyzed the effect of code transformation on clone detection accuracy. We applied a number of code normalization and abstractions equipped with NiCad. Results indicated that code transformation enables detecting extra near-miss and semantic clones, but produces more false positives. Byte code represents a well-normalized form for the source code. Since Microsoft .NET produces a unified intermediate language for all its programming languages, we used it to detect clones across Microsoft .NET languages (Chapter 4). In this chapter, we analyzed the intermediate language in-depth by applying different filters that enhance the accuracy of identifying cross-language clones. The result has shown the ability of intermediate code in detecting cross-language clones. The literature defines cross-language clones as a pair of code fragments that performs the same functionality implemented in two different languages. This definition is close to the semantic clone definition. We further

conducted a study using CIL for the detection of semantic clones (Chapter 5). We extracted more features from the byte code after applying the same filters, then we used different matching algorithms for different features. Our approach enabled the detection of more clones. Precision evaluation performed manually by validating a random sample from our tool (NetSim) and comparing it to other tools. But, recall validation was difficult because there is no specialized semantic clone benchmark. Also, NetSim does not support languages in the currently available benchmarks.

To help evaluate semantic clone detectors, we built a semantic clone benchmark (Chapter 6). In this study, we utilized crowd-sourced knowledge to extract semantic clones from Stack Overflow. Clones are extracted from correct programming answers on Stack Overflow. Extraction involved several steps to ensure the syntax and semantic correctness of potential candidates. Before manual validation, we filtered out our syntax clones that are detected by other tools to make sure the benchmark has real semantic clones. Finally, we used the semantic clone benchmark to evaluate and compare the semantic clone detection tools (Chapter 7). We developed an injection benchmark that takes a clone from a benchmark, injects it into a subject system, runs the detection tool to be evaluated, and finally evaluates the tool's recall. We then analyzed semantic clones detected by detection tools and available semantic clones in benchmarks to build a taxonomy for semantic clones that we used to create a second semantic clone benchmark. We also used this benchmark to evaluate detection tools.

8.2 Contributions

The work in this thesis aims to contribute to the state of the art in the semantic code clone area as follows:

Assessing the cloning status in games and across games. In this study, we used NiCad to detect clones in games, VisCad to visualize results, and measure many cloning metrics to understand the cloning level in and across games. The main finding is that games have fewer clones and there is no evidence of copied code across games.

Assessing the effect of code transformation in clone detection. This study investigates code filtering, normalization, and abstraction options that came with NiCad in detecting clones.

Cross-language detection technique. A technique to detect clones across .NET programming languages that detects clones at the level of byte code. The byte code goes through several filtering steps that reduces processing time and increases accuracy.

Semantic clone detection technique. This technique is based on byte code and detects clones at the method level in .NET programming languages.

SemanticCloneBench. A semantic clone benchmark that consists of four thousand real semantic clones in four programming languages, C, C#, Java, and Python. It is the only specialized benchmark for semantic clones for evaluating semantic clone detection tools.

Semantic clone taxonomy. A comprehensive semantic clone taxonomy that discusses different types of semantic clones. The taxonomy provides an example for each type, which helps better define the definition of semantic clones.

Evaluating and comparing the recall of semantic clone detection tools. We developed an injection framework that automatically measures the recall for clone detection tools. Using SemanticCloneBench we measure the recall for syntax and semantic detection tools. Also, we measure and compare the precision and execution time for the tools.

8.3 Future Research Directions

In this thesis, we addressed a number of issues related to semantic code clones, including detection, benchmarking, and evaluating detection tools. Identifying identical functionality (semantic clones) in source code is undecidable; semantic clones can result from unlimited implementations (i.e., different algorithms and data structures). A high accuracy semantic clone detection tool would open up semantic clone research into a variety of areas, such as analysis, refactoring, patterns, and harmfulness. The following semantic clone research challenges remain: creating a high precision, recall, and scalable semantic clone detector; and creating a benchmark for all clone types and for different programming languages. As well, a number of empirical studies could be considered, such as finding and analyzing patterns of semantic clones, their types, and existence; exploring the relationship between semantic clones and system performance; and, investigating the importance of semantic clones for system quality, maintenance, and comprehension (e.g., the positive and negative effects of semantic clones and whether they should be refactored or kept in the system). As well, our current work could be extended in a variety of ways, including the following areas.

Cross language clone detection

A study to investigate the use of other intermediate representations found in other frameworks, such as the unified bytecode generated by LLVM compiler [6] to detect clones across LLVM programming languages. Moreover, we are going to start a comprehensive usability study using our clone detection approach in an enterprise software development environment to observe the unexplored characteristics of multi-language clone detection in software maintenance process.

Expansion of SemanticCloneBench

For the next release of the semantic clone benchmark, we plan to extend the benchmark with other languages and double validate the clones using more judges. Also, we could extend the benchmark with syntactic clones that are extracted from Stack Overflow using a similar procedure. In addition, there is some interesting future work that we plan to conduct. For example, we will use the benchmark to evaluate more semantic clone detectors and syntax clone detectors.

Source code transformation and clone detection

An empirical study could be taken to study the effects of more code normalization and transformation. Our taxonomy could be used to apply different code transformations, where each type in the taxonomy could be used to define a code transformation. All taxonomy types that improve semantic clone detection could be combined in a final semantic clone detector.

Automatic semantic clone creation

We used the taxonomy to create 60 clones in three programming languages. We aim to automate creating semantic clones according to the definition provided by the taxonomy. A combination of taxonomy types could be used to create more complex semantic clones. This enables us to create more clones to evaluate detection tools. In future work, we plan to evaluate more semantic clone detection tools.

Semantic clone detection using deep learning

Our semantic clone benchmark represents a good training data set for machine learning and deep learning algorithms. We believe that our benchmark will work well with machine learning to detect semantic clones.

Evaluating more aspects of clone detection tools

Other aspects related to clone detection and semantic clone detection tools could be evaluated. Another empirical study could be performed to investigate how target system size affects the accuracy of clone detection tools. Also, we could use our injection framework to analyse how injection location affects the ability of tools to detect clones.

8.4 Summary

Code cloning is an active research area. More than a hundred clone detection tools and techniques have been proposed. These tools are used in research related to code analysis, understanding, refactoring, maintenance, and management. Many factors govern tool selection for a specific application, such as the programming language, the system size, the clone types, and the tool's accuracy and speed. Clone related studies target all types of clones, but less research has been conducted on semantic clones.

In this thesis, we advanced the state-of-the-art for semantic clones. We contributed to cloning analysis by studying cloning in game systems and analyzing the effect of code normalization in detecting semantic clones. We contributed to semantic clone detection with a semantic clone detection tool (NetSim) and cross-language clone detection technique. We also contributed to the area of clone benchmarking and evaluation with a semantic clone benchmark (SemanticCloneBench) for measuring the semantic recall of semantic clone detection tools. Finally, our semantic clone taxonomy supports better understanding of semantic clones with examples. Results and experiments in this thesis open new directions for future work. Our contributions lead to possible future work that could contribute to semantic clone analysis, detection, benchmarking, and evaluation.

References

[1] Asxgui-video encoder gui. https://sourceforge.net/projects/asxgui/. Accessed: Sep, 2020.

[2] Id software. http://www.idsoftware.com/. Accessed: March, 2021.

[3] Itext dot net package. http://www.ujihara.jp/iTextdotNET/en/index.html. Accessed: Sep, 2017.

[4] Itext pdf library. https://itextpdf.com/en. Accessed: Sep, 2020.

[5] Jimple: Simplifying java bytecode for analyses and transformations. https://www.sable.mcgill.ca/soot/doc/soot/jimple/Jimple.html. Accessed: Jan, 2021.

[6] The llvm compiler infrastructure. http://www.llvm.org/. Accessed: Sep, 2020.

[7] Mono: Cross platform, open source .net framework. https://www.mono-project.com/. Accessed: Sep, 2020.

[8] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. On code reuse from stackoverflow : An exploratory study on android apps. *Information and Software Technology*, 88, 04 2017.

[9] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE Access*, 7:86121–86144, 2019.

[10] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–, Nov 2005.

[11] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling. Detecting clones across microsoft .net programming languages. In *2012 19th Working Conference on Reverse Engineering*, pages 405–414, Oct 2012.

[12] F. Al-Omari, C. K. Roy, and T. Chen. Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 57–63, 2020.

[13] Farouq Al-Omari and Chanchal Roy. Is code cloning in games really different? In *Proceedings of the 31st Annual ACM Symposium on applied computing*, volume 04-08- of *SAC '16*, pages 1512–1519. ACM, 2016.

[14] Hakam W. Alomari and Matthew Stephan. Srcclone: Detecting code clones via decompositional slicing. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 274–284. Association for Computing Machinery, 2020.

[15] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. Stack overflow: A code laundering platform? *CoRR*, abs/1703.03897, 2017.

[16] Christian Arwin and S. M. M. Tahaghoghi. Plagiarism detection across programming languages. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, pages 277–286, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[17] Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. Viscad: Flexible code clone analysis support for nicad. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 77–78, New York, NY, USA, 2011. ACM.

[18] Franz Baader, Diego Calvanese, Deborah Mcguinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 01 2007.

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

[20] Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 15–15, Berkeley, CA, USA, 1998. USENIX Association.

[21] Sebastian Baltes, Christoph Treude, and Stephan Diehl. Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR 2019)*, 2019.

[22] Hamid Abdul Basit, Usman Ali, and Stan Jarzabek. Viewing simple clones from structural clones' perspective. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, page 1–6, New York, NY, USA, 2011. Association for Computing Machinery.

[23] Saman Bazrafshan. No clones, no trouble? In *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pages 37–38, Piscataway, NJ, USA, 2013. IEEE Press.

[24] 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, September 2007.

[25] P. M. Caldeira, K. Sakamoto, H. Washizaki, Y. Fukazawa, and T. Shimada. Improving syntactical clone detection methods through the use of an intermediate representation. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 8–14, 2020.

[26] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 380–388. Association for Computing Machinery, 2002.

[27] Long Chen, Wei Ye, and Shikun Zhang. Capturing source code semantics via tree-based convolution over api-enhanced ast. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, page 174–182, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Y. Chen, I. Keivanloo, and C. K. Roy. Near-miss software clones in open source games: An empirical study. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–7, May 2014.

[29] Xiao CHENG, Zhiming PENG, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. Clcminer: Detecting cross-language clones without intermediates. *IEICE Transactions on Information and Systems*, E100.D:273–284, Feb 2017.

[30] Herman Chernoff. The use of faces to represent points in k-dimensional space graphically. *Journal of the American Statistical Association*, 68(342):361–368, 1973.

[31] M. C. Chuah and S. G. Eick. Glyphs for software visualization. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*, pages 183–191, 1997.

[32] Paolo Ciancarini and Gian Piero Favini. Plagiarism detection in game-playing software. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, page 264–271, New York, NY, USA, 2009. Association for Computing Machinery.

[33] James R. Cordy. Comprehending reality " practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 196–, Washington, DC, USA, 2003. IEEE Computer Society.

[34] James R. Cordy. Source transformation, analysis and generation in txl. In *PEPM*, 2006.

[35] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Transactions on Mobile Computing*, 14(10):2007–2019, Oct 2015.

[36] Antonio Cuomo, Antonella Santone, and Umberto Villano. A novel approach based on formal methods for clone detection. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 8–14. IEEE, Jun 2012.

[37] Olivier Cur and Guillaume Blin. *RDF Database Systems: Triples Storage and SPARQL Query Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[38] Tim A D. Henderson and Andy Podgurski. Rethinking dependence clones. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE, Feb 2017.

[39] Neil Davey, Paul Barson, Simon Field, Ray Frank, and Stewart Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1, Jan 1995.

[40] Ian J. Davis and Michael W. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *2010 17th Working Conference on Reverse Engineering*, pages 242–246. IEEE, Oct 2010.

[41] F Deissenboeck, M Pizka, and T Seifert. Tool support for continuous quality assessment. In *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pages 127–136. IEEE, 2005.

[42] Florian Deissenboeck, Benjamin Hummel, and Elmar Jürgens. Code clone detection in practice. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, pages 499–500, 01 2010.

[43] Ekwa Duala-Ekoko and Martin P. Robillard. Clonetracker: Tool support for code clone management. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 843–846, New York, NY, USA, 2008. ACM.

[44] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–118, Washington, DC, USA, 1999. IEEE Computer Society.

[45] Abdulrahman Abu Elkhail, Jan Svacina, and Tomas Cerny. Intelligent token-based code clone detection system for large scale source code. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, RACS '19, page 256–260, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Rochelle Elva. *Detecting Semantic Method Clones In Java Code Using Method Ioe-behavior*. PhD thesis, University of Central Florida, 2013.

[47] Rochelle Elva and Gary T. Leavens. Semantic clone detection using method ioe-behavior. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 80–81, Piscataway, NJ, USA, 2012. IEEE Press.

[48] William S. Evans, Christopher W. Fraser, and Fei Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, December 2009.

[49] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 516–527, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Fang-Hsiang Su, J. Bell, G. Kaiser, and S. Sethumadhavan. Identifying functionally similar code in complex codebases. In *ICPC*, pages 1–10, May 2016.

[51] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? the impact of copy paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, May 2017.

[52] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[53] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[54] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 147–156, New York, NY, USA, 2010. Association for Computing Machinery.

[55] Neelamadhav Gantayat, Pankaj Dhoolia, Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. The synergy between voting and acceptance of answers on stackoverflow, or the lack thereof. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 406–409, Piscataway, NJ, USA, 2015. IEEE Press.

[56] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang, and Y. Cai. Teccd: A tree embedding approach for code clone detection. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 145–156, 2019.

[57] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, March 2006.

[58] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, FASE'06, pages 411–425, Berlin, Heidelberg, 2006. Springer-Verlag.

[59] Nils Gode and Jan Harder. Clone Stability. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 65–74. IEEE, March 2011.

[60] Torsten Gorg. Deriving categories of semantic clones from a coding contest. In *Softwaretechnik-Trends: Vol. 37, No. 2*, pages 58–59, Berlin, 2017. Gesellschaft für Informatik e.V., Fachgruppe PARS.

[61] Stefan Haefliger, Georg von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, January 2008.

[62] Jan Harder and Nils Göde. Cloned code: Stable code. *Journal of Software: Evolution and Process*, 25:1063–1088, Oct 2013.

[63] Simon Harris. Simian - similarity analyser. `https://www.harukizaemon.com/simian/`. Accessed: Sep, 2020.

[64] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. Teamscale: software quality control in real-time. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 592–595. ACM Press, 2014.

[65] Yoshiki Higo and Shinji Kusumoto. Code clone detection on specialized PDGs with heuristics. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 75–84, Washington, DC, USA, 2011. IEEE Computer Society.

[66] 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, oct 2011.

[67] Wiebe Hordijk, María Laura Ponisio, and Roel Wieringa. Harmfulness of code duplication: a structured review of the evidence. *Journal of Signal Processing Systems - JSPS*, pages 88–97, April 2009.

[68] Keisuke Hotta, Yui Sasaki, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. An empirical study on the impact of duplicate code. *Advances in Software Engineering*, 2012:1–22, January 2012.

[69] Y. Hu, G. Xu, B. Zhang, K. Lai, G. Xu, and M. Zhang. Robust app clone detection based on similarity of ui structure. *IEEE Access*, 8:77142–77155, 2020.

[70] James W. Hunt and M. Douglas McIlroy. An algorithm for differential file comparison. Technical Report 9, Stanford University, Bell Laboratories, Murray Hill, New Jersey 07974, 1976.

[71] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.

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

[73] Paul Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bulletin de la Societe Vaudoise des Sciences Naturelles*, 37:241–72, 01 1901.

[74] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 96–105, USA, 2007. IEEE Computer Society.

[75] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, New York, NY, USA, 2009. ACM.

[76] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, page 171–183. IBM Press, 1993.

[77] E Juergens, F Deissenboeck, and B Hummel. Code Similarities Beyond Copy & Paste. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 78–87. IEEE, Mar 2010.

[78] V. Juričić. Detecting source code similarity using low-level languages. In *Proceedings of the ITI 2011, 33rd International Conference on Information Technology Interfaces*, pages 597–602, June 2011.

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

[80] Toshihiro Kamiya. Agec: An execution-semantic clone detection tool. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 227–229. IEEE, May 2013.

[81] Toshihiro Kamiya. An execution-semantic and content-and-context-based code-clone detection and analysis. In *2015 IEEE 9th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE, Mar 2015.

[82] 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, July 2002.

[83] Cory Kapser and Michael Godfrey. Supporting the analysis of clones in software systems: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, Jan 2006.

[84] Cory Kapser and Michael W. Godfrey. Toward a taxonomy of clones in source code: A case study, 2003.

[85] Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pages 85–94, Sep. 2004.

[86] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, December 2008.

[87] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 36–42, Piscataway, NJ, USA, 2012. IEEE Press.

[88] M. Kessel and C. Atkinson. On the efficacy of dynamic behavior comparison for judging functional equivalence. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 193–203, 2019.

[89] Byoungchul Kim, Kyeonghwan Lim, Seong-Je Cho, and Minkyu Park. Romadroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app's manifest file. *IEEE Access*, PP:72182–72196, May 2019.

[90] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[91] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, September 2005.

[92] R Komondoor and S Horwitz. Tool demonstration: Finding duplicated code using program dependences. *European Symposium on Programming*, 2028:383–386, 2001.

[93] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, 2001. Springer-Verlag.

[94] K A Kontogiannis, R Demori, E Merlo, M Galler, and M Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, Jun 1996.

[95] Kostas Kontogiannis, Panos Linos, and Kenny Wong. Comprehension and Maintenance of Large-Scale Multi-Language Software Applications. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 497–500. IEEE, sep 2006.

[96] Nicholas Kraft, Brandon Bonds, and Randy Smith. Cross-language clone detection. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE'08)*, pages 54–59, Jan 2008.

[97] Ronald Krawitz. *Code Clone Discovery Based on Functional Behavior*. PhD thesis, Nova Southeastern University, 2012. CCE Theses and Dissertations.

[98] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–309, Washington, DC, USA, 2001. IEEE Computer Society.

[99] Jens Krinke. Is Cloned Code More Stable than Non-cloned Code? In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 57–66. IEEE, Sep 2008.

[100] D. E. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 489–490, Oct 2013.

[101] Daniel E. Krutz and Wei Le. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 388–391, New York, NY, USA, 2014. ACM.

[102] Daniel Edward Krutz. *Code Clone Discovery Based on Concolic Analysis*. PhD thesis, Nova Southeastern University, 2013. AAI3554681.

[103] V. Käfer, S. Wagner, and R. Koschke. Are there functionally similar code clones in practice? In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, pages 2–8, March 2018.

[104] Da-Young Lee, Uram Ko, Ibrahim Aitkazin, SangUn Park, Hae-Sung Tak, and Hwan-Gue Cho. A fast detecting method for clone functions using global alignment of token sequences. In *Proceedings of the 2020 12th International Conference on Machine Learning and Computing*, ICMLC 2020, page 17–22, New York, NY, USA, 2020. Association for Computing Machinery.

[105] António Menezes Leitão. Detection of redundant code using r2d2. *Software Quality Journal*, 12(4):361–382, December 2004.

[106] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[107] B. Li, C. Ye, S. Guan, and H. Zhou. Semantic code clone detection via event embedding tree and gat network. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 382–393, 2020.

[108] G. Li, Y. Wu, C. K. Roy, J. Sun, X. Peng, N. Zhan, 33 B. Hu, and J. Ma. Saga: Efficient and large-scale detection of near-miss clones with gpu acceleration. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 272–283, 2020.

[109] 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*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

[110] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Analysis of the linux kernel evolution using code clone coverage. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 22–25, Washington, DC, USA, 2007. IEEE Computer Society.

[111] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *2008 IEEE International Conference on Software Maintenance*, pages 227–236, Sep. 2008.

[112] Angela Lozano and Michel Wermelinger. Tracking clones' imprint. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, page 65–72, New York, NY, USA, 2010. Association for Computing Machinery.

[113] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 18–, Washington, DC, USA, May 2007. IEEE Computer Society.

[114] Nikolay Mateev, Vijay Menon, and Keshav Pingali. Fractal symbolic analysis. In *Proceedings of the 15th international conference on Supercomputing - ICS '01*, pages 38–49, New York, New York, USA, 2001. ACM Press.

[115] George Mathew, Chris Parnin, and Kathryn T Stolee. Slacc: Simion-based language agnostic code clones. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 210–221, New York, NY, USA, 2020. Association for Computing Machinery.

[116] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 244–253, Washington, DC, USA, 1996. IEEE Computer Society.

[117] Audris Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, page 7, USA, 2007. IEEE Computer Society.

[118] Debajyoti Mondal, Manishankar Mondal, Chanchal K. Roy, Kevin A. Schneider, Yukun Li, and Shisong Wang. Clone-world: A visual analytic system for large scale software clones. *Visual Informatics*, 3(1):18–26, 2019. Proceedings of PacificVAST 2019.

[119] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider. An empirical study of the impacts of clones in software maintenance. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 242–245, June 2011.

[120] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. Associating code clones with association rules for change impact analysis. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 93–103, 2020.

[121] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on bug propagation through code cloning. *Journal of Systems and Software*, 158:110407, 2019.

[122] Manishankar Mondal, Chanchal K. Roy, Md. Saidur Rahman, Ripon K. Saha, Jens Krinke, and Kevin A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1227–1234, New York, NY, USA, 2012.

[123] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on clone stability. *ACM SIGAPP Applied Computing Review*, 12(3):20–36, September 2012.

[124] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. A survey on clone refactoring and tracking. *Journal of Systems and Software*, 159:110429, 2020.

[125] Golam Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Clonecognition: Machine learning based code clone validation tool. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1105–1109. Association for Computing Machinery, 2019.

[126] J. Munro, C. Boldyreff, and A. Capiluppi. Architectural studies of games engines — the quake series. In *2009 International IEEE Consumer Electronics Society's Games Innovations Conference*, pages 246–255, Aug 2009.

[127] Kawser Wazed Nafi, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. A universal cross language software similarity detector for open source software categorization. *Journal of Systems and Software*, 162, 2020.

[128] Lawton Nichols, Mehmet Emre, and Ben Hardekopf. Structural and nominal cross-language clone detection. In Reiner Hähnle and Wil van der Aalst, editors, *Fundamental Approaches to Software Engineering*, pages 247–263, Cham, 2019. Springer International Publishing.

[129] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 518–528. IEEE Press, 2019.

[130] Bayu Priyambadha and Siti Rochimah. Case study on semantic clone detection based on code behavior. In *2014 International Conference on Data and Software Engineering (ICODSE)*, pages 1–6. IEEE, Nov 2014.

[131] Jing Qiu, Xiaohong Su, and Peijun Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering*, 42(2):187–202, Feb 2016.

[132] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixão, Giuseppe Bianco, and Rocco Oliveto. Toxic code snippets on stack overflow. *CoRR*, abs/1806.07659, 2018.

[133] M. S. Rahman and C. K. Roy. A change-type based empirical study on the stability of cloned code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 31–40, Sep. 2014.

[134] Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web applications. In *Proceedings of the 5th International Conference on Web Engineering*, ICWE'05, pages 252–262, Berlin, Heidelberg, 2005. Springer-Verlag.

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

[136] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. *Journal of Software Maintenance and Evolution: Research and Practice.*, 22(3):165–189, April 2010.

[137] C. K. Roy, M. F. Zibran, and R. 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, Feb 2014.

[138] Chanchal K. Roy and James R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, Jun 2008. IEEE Computer Society.

[139] 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.

[140] 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, May 2009.

[141] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *School of Computing TR 2007-541, Queen's University*, 541:115, 2007.

[142] 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*, ESEC/FSE 2018, pages 354–365, New York, NY, USA, 2018. ACM.

[143] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Di Yang, Pedro Martins, Hitesh Sajnani, Pierre Baldi, and Cristina V. Lopes. Towards automating precision studies of clone detectors. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 49–59. IEEE Press, 2019.

[144] 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*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[145] Sandro Schulze, Sven Apel, and Christian Kästner. Code clones in feature-oriented software product lines. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, page 103–112, New York, NY, USA, 2010. Association for Computing Machinery.

[146] Gehan M.K. Selim, King Chun Foo, and Ying Zou. Enhancing source-based clone detection using intermediate representation. In *2010 17th Working Conference on Reverse Engineering*, pages 227–236. IEEE, Oct 2010.

[147] S. Shahzad, A. Hussain, and S. Nazir. A clone management framework to improve code quality of foss projects. In *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*, pages 253–258, 2017.

[148] Abdullah Sheneamer and Jugal Kalita. Semantic Clone Detection Using Machine Learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024–1028. IEEE, Dec 2016.

[149] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 702–714, New York, NY, USA, 2016. ACM.

[150] M. Suzuki, A. C. d. Paula, E. Guerra, C. V. Lopes, and O. A. L. Lemos. An exploratory study of functional redundancy in code repositories. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 31–40, Sep. 2017.

[151] Jan Svacina, Jonathan Simmons, and Tomas Cerny. Semantic code clone detection for enterprise applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 129–131. Association for Computing Machinery, 2020.

[152] Jeffrey Svajlenko. *Large-Scale Clone Detection and Benchmarking.* PhD dissertation, University of Saskatchewan, December 2017.

[153] Jeffrey Svajlenko, Judith F. Islam, Iman , Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 476–480, Washington, DC, USA, 2014. IEEE Computer Society.

[154] Jeffrey Svajlenko and Chanchai K. Roy. Fast, scalable and user-guided clone detection. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 352–353, New York, NY, USA, 2018. ACM press.

[155] Jeffrey Svajlenko and Chanchal Roy. The mutation and injection framework: Evaluating clone detection tools with mutation analysis. *IEEE Transactions on Software Engineering*, PP:1–1, 04 2019.

[156] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating modern clone detection tools. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 321–330, Washington, DC, USA, 2014. IEEE Computer Society.

[157] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 131–140, Washington, DC, USA, 2015. IEEE Computer Society.

[158] Jeffrey Svajlenko and Chanchal K. Roy. Fast and flexible large-scale clone detection with cloneworks. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, page 27–30. IEEE Press, 2017.

[159] Jeffrey Svajlenko and Chanchal K. Roy. A survey on the evaluation of clone detection performance and benchmarking, 2020.

[160] Jeffrey Svajlenko and Chanchal Kumar Roy. Efficiently measuring an accurate and generalized clone detection precision using clone clustering. In *SEKE*, 2016.

[161] Ryo Tajima, Masataka Nagura, and Shingo Takada. Detecting functionally similar code within the same project. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, pages 51–57. IEEE, Mar 2018.

[162] H. Thaller, L. Linsbauer, and A. Egyed. Towards semantic clone detection via probabilistic software modeling. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 64–69, 2020.

[163] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, page 173–180, USA, 2004. IEEE Computer Society.

[164] M. S. Uddin, C. K. Roy, and K. A. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 236–238, May 2013.

[165] Md. Sharif Uddin, Chanchal K. Roy, Kevin A. Schneider, and Abram Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

[166] B. van Bladel and S. Demeyer. A novel approach for detecting type-iv clones in test code. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 8–12, 2019.

[167] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. How are functionally similar code clones syntactically different? an empirical study and a benchmark. *PeerJ Computer Science*, 2:e49, Mar 2016.

[168] Andrew Walenstein, Nitin Jyoti, Junwei Li, Junwei Li, Yun Yang, and Arun Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 285–294, Washington, DC, USA, 2003. IEEE Computer Society.

[169] S. Wang, T. P. Chen, and A. E. Hassan. How do users revise answers on technical q a websites? a case study on stack overflow. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[170] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin. Scdetector: Software functional clone detection based on semantic tokens analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 821–833, 2020.

[171] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Clonediff: Semantic differencing of clones. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 83–84, New York, NY, USA, 2011. ACM.

[172] D. Yang, P. Martins, V. Saini, and C. Lopes. Stack overflow in github: Any snippets there? In *MSR '17*, pages 280–290, May 2017.

[173] Shunsuke Yoshioka, Norihiro Yoshida, Kyohei Fushida, and Hajimu Iida. Scalable Detection of Semantic Clones Based on Two-Stage Clustering. International Symposium on Software Reliability Engineering (ISSRE 2011), 2011.

[174] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 70–80. IEEE Press, 2019.

[175] Y. Yuan, W. Kong, G. Hou, Y. Hu, M. Watanabe, and A. Fukuda. From local to global semantic clone detection. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, pages 13–24, 2020.

[176] Yusuke Yuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Generating clone references with less human subjectivity. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4. IEEE, May 2016.

[177] Fanlong Zhang and Siau cheng Khoo. An empirical study on clone consistency prediction based on machine learning. *Information and Software Technology*, 136:106573, 2021.

[178] Y. Zou, B. Ban, Y. Xue, and Y. Xu. CCGraph: a PDG-based code clone detector with approximate graph matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–942, 2020.