Summary

The thesis was written about searching semantic code clones using the NiCad clone detector on a stubbed BigCloneBench. The NiCad clone detector has a plug-in architecture that is extendable using the programming language TXL.

The research focuses on testing whether it was possible to use a code clone detector NiCad to find source clones using JVM Bytecode or Jimple Registry Code intermediary representation. Suppose it is proven that NiCad could detect type-3 and type-4 clones within these intermediary representations. In that case, it is undoubtedly possible to detect semantic clones for programming languages that can be transformed into JVM Bytecode or Jimple Register Code.

The dataset BigCloneBench was used and then preprocessed into a readable version of it in JVM Bytecode form. Due to the change of structure, the detected clones need to be mapped back to their source code metadata. BigCloneEval was then used after the metadata were remapped. The metadata required by BigCloneEval are the startline, endline, filename, and subfolder name for each clone fragment.

NiCad was extended to take JVM Bytecode as input. The experiment was

done on an evaluation tool BigCloneEval on the stubbed BigCloneBench, which was transformed into its JVM Bytecode counterpart. The result was that we were able to find almost all type-1 and type-2 clones and relatively high type-3 and type-4 clones depending on the threshold configuration in NiCad.

Zusammenfassung

Diese Arbeit zielt darauf ab, die Forschungsfrage zu beantworten, ob es möglich ist, Quellklone in NiCad unter Verwendung von Java Bytecode und Jimple Register Code zu erkennen. NiCad wird durch eine Plug-in-Architektur erweitert, die die funktionale Sprache TXL verwendet. Die Experimente werden mit dem Datensatz BigCloneBench durchgeführt, der mit Stubber gestubbt wird. Der Datensatz wird in eine lesbare JVM-Bytecode-Form vorverarbeitet und ein Mapper zwischen den Original-Metadaten und den Metadaten in der JVM-Bytecode-Form wurde erstellt. Dieser Mapper wurde verwendet, weil wir BigCloneEval als Evaluierungswerkzeug für unseren Klon-Detektor eingesetzt haben. BigCloneEval wurde für BigCloneBench entwickelt. Es wurde gezeigt, dass es tatsächlich möglich ist, auch mit relativ hoher Präzision Quellcode-Klone zu finden.

Abstract

This thesis aims to answer the research question of whether it is possible to detect source clones in NiCad using Java Bytecode and Jimple Register Code. NiCad is extended using a plug-in architecture, utilizing the functional language TXL. The experiments are carried out using the dataset BigCloneBench which is stubbed using Stubber. The dataset is preprocessed into a readable JVM Bytecode form and a mapper between the original metadata and its JVM Bytecode form metadata was made. This mapper was used because we used BigCloneEval as an evaluation tool for our clone detector. BigCloneEval was made for BigCloneBench. It was shown that it was actually possible, also with relatively high precision to find source code clones.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Background Problem

Code clone detection is an important tool for software engineering [1]. It has many uses such as analyzing redundancy within a code base to facilitate refactoring. Plagiarism and security vulnerability detection are other huge factors for which it can be used. Various factors could lead to the production of code clones. This may include code reuse, lack of communication, and inconsistent coding practices. Code clones will inevitably be created as a byproduct of a large project. There are disadvantages to having code clones, which will significantly influence the cost of maintaining the code base due to possible bug propagation [2]. In some cases, it is also possible to detect code clones for web pages to reduce testing effort [3].

Clone detection is a very complicated topic. It requires different techniques

and algorithms to perform [4]. There are various clone detectors, such as Komondoor [5], SourcererCC, and Duplo [6]. In this thesis, the clone detector NiCad [1] was used. The NiCad clone detector will be used to check for source code clones within BigCloneBench [7], which was transformed into Java Bytecode [8] and Jimple [9] Register Code beforehand.

1.2 Scope and Limitation of the Thesis

The thesis used the BigCloneBench dataset, which was compilable without dependencies. To achieve this goal, Stubber [10] was used. Stubber is capable of compiling Java source code into Bytecode without dependencies. There have already been researches where the Java Bytecode was compiled and Java source code clone detectors were used [11]. Jimple [9] register code will also be created using the Bytecode to see the result of the clone detection on a register code.

NiCad [12] was used to find the code clones. The result was then processed and evaluated on BigCloneEval [13]. The recall of both methods was then measured based on the report BigCloneEval produced.

However, this approach is not always viable due to obfuscation [14]. The experiment was restricted to NiCad and will not reflect on the potential

results produced by other clone detectors. Therefore, it was also limited by the performance of NiCad due to its dependency on the syntactic structure of the code, which could miss potential semantic clones.

1.3 Definition of Terms

Java Bytecode is an intermediate representation of Java created by the compiler. It is also used by various other languages such as Clojure, Scala, and Kotlin. It is compiled from a Java compiler and produced as .class files. It allows portability for the code, enabling it to be written once and be usable on different platforms. One example usage of this is that it can be used for detecting code clones within an Android app [15]. It is possible to do dynamic loading and linking of classes, to improve modularity and extensibility.

Soot framework is applied on Java Bytecode to construct Jimple. Jimple is an intermediate representation that can be used to analyze Java Bytecode. It utilizes a typed, three-address implementation that makes it easier to read and understand. The instructions are mostly three operands with registers, constants, and variables.

Within a code base, there exists code that is copied and reused with minor or major modifications, this is a common practice in engineering. It is also

```
int func(int[] arr) {
    int s = 0;
    for (int i = 0; i<arr.length; i++)
        if (arr[i]>0)
            s *= (arr[i] * fu());
    return s;
}
```

(a) Source code

```
push 0
store.i i5
push 0
store.i i6
load.i i6
load.r r0
arraylength
ifcmppge.j goto 24
load.r r0
load.i i6
arrayread.i
ifle goto 22
load.i i5
load.r r0
load.i i6
arrayread.i
load.r r1
virtualinvoke fu()
mul.i
mul.i
store.i i5
inc.i i6 1
goto 5
load.i i5
return.i
```

(b) Bytecode Equivalent

```
i5 = 0
i6 = 0
$i0 = lengthof r0
if i6 >= $i0 goto 13
$i1 = r0[i6]
if $i1 <= 0 goto 11
$i2 = r0[i6]
$i3 = virtualinvoke fu()
$i4 = $i2 * $i3
i5 = i5 * $i4
i6 = i6 + 1
goto 3
return i5
```

(c) Jimple Register Code Equivalent

Figure 1.1: An example of a Java source code and their Java Bytecode and Jimple register code equivalent [16]

very possible to be created by accident in larger projects, mostly due to a lack of communication between project members. Codes that have similar syntax and or semantic meaning are categorized as code clones. There are various advantages and disadvantages of having code clones. Clone detection

is essential for several tasks such as plagiarism or virus detection, or code quality analysis [4]. Having code clones can have disadvantages, such as obstructing bug fixes due to its propagating property [17], especially due to the source code cloned having bugs.

There are different types of code clones [18]. The ones that are relevant to this thesis are Type-1, Type-2, Type-3, and Type-4 code clones [17]. Type-1 code clones are exact clones with 100% similarity [19], and type-2 code clones are the ones where the names of the functions and variables are anonymized. Type-3 near-miss code clones may contain several instructions altered, added, or deleted. Type-4 semantic clone has a very different syntax but computes the same tasks. It is also possible for type-4 semantic clones to occur cross-languages [20].

1.4 Approach of the Thesis

This thesis consists of several steps. First, it will take Java files from BigCloneBench. Stubber then filtered out the Java files. It will then be transformed into .bc files, which are readable .class Java Bytecode files. The .bc files are used as an input to NiCad and then the output will be code clones in XML form. The code clones in the XML form will then be unwrapped and fitted into the input format for BigCloneEval using a script in Bytecode

Texelite. A script from Stubber will be used once again to change the start line and end line of the code clones, to match the BigCloneBench that is used with Stubber.

1.5 Aim of this Thesis

Many languages compile into an intermediary representation known as Java Bytecode [8]. This thesis aims to extend the NiCad Clone Detector to also process Java Bytecode files in the form of .bc and registry code. This potentially enables a much bigger coverage of the codes that can be analyzed, without creating a code clone detector for each language that compiles into Java Bytecode.

2 Related Work

It is ubiquitous in practice to copy and paste code fragments within the field of software engineering [21]. Such practice would inherently produce code clones. According to research, a large portion of the code in a software system is cloned [17].

2.1 Code Clone Detection

Code clone detection is an ever-expanding field. Various applications can be associated with code clone detection. According to studies, almost half of the large software systems consist of code clones [6, 22]. With it, there will also be various questions that must be solved. The complexity of the detection algorithm, the tool limitations, human factors, and organizational challenges are all significant topics that must be addressed in code clone detection.

Code clone detection approaches can be categorized into six categories based on their method [2]:

- textual-based
- lexical-based
- tree-based
- semantic-based
- metric-based
- hybrid

textual-based code clone detection is done by reading the text directly contained within the program files. It is the best at detecting type 1 clones, however, it is also capable of detecting type 2, type 3, and type 4 clones as well [2].

The lexical approach on the other hand converts the source code into tokens. These sequences of tokens are then compared to each other. It has the advantage of being language agnostic due to it being token-based. A well-known example of a lexical-based code clone detector would be CCFinder [23] (later replaced by CCFinderX [24]) and ScalClone [25].

Tree-based methods in code clone detection focus more on comparing the structure of the clones, rather than the exact content itself. Abstract Syntax Tree obtained through parsing is commonly used in this approach as the representation of the program to be compared to one another [2, 26]. An example of an established code clone detector that has a tree-based approach would be DECKARD [27]. Tree-based clone detectors are, however, more computationally intensive, especially for larger codebases.

The semantic-based approach analyzes the code clones through the semantic meaning of the code itself, by contrast to observing only its syntactic structure or lexical sequences. It is mainly used to discover type-4 clones [28]. It is capable of cross-language clone detection due to the languages being converted to a high-level representation. This is useful due to clones being made cross-language within a large system [20]. An example of a semantic-based approach would be the Language Independent Code Clone Analyzer (LICCA) [28]. It uses the source code in the form of tree-based intermediate representation through the Set of Software Quality Static Analyser [29]. The source code is saved in the form of an enriched Concrete Syntax Tree, containing concrete tokens that are able to be reconstructed back into the source code. The enriched Concrete Syntax Tree is created through the Set of Software Quality Static Analyser eCSTGenerator. This enriched Concrete

Syntax Tree can store semantic information between the source codes from different programming languages, which will be used as the input to LICCA.

The metric-based approach to code clone detection works by utilizing metrics to calculate the similarity between the clones [2]. Usually, the source code will be transformed into abstract syntax trees or program dependency graphs and then compared using a comparison metric. An example of a metric-based code clone detector would be Clone Works [30]. The clone detection will be executed through a metric based on the Jaccard similarity metric [31] using the clone fragments as input.

Hybrid-based approach, on the other hand, combines two or more of the various code clone detection methods. It generally outperforms the other singular code clone detection approaches.

Code clone detections are conducted by using various tools. Examples of commonly used tools are NiCad [1, 12], CPMiner [32], CCFinderX [24] and others. Moreover, there are also a variety of algorithms used for code clone detection, such as using a Neural Network [33] or using the Euclidian Distance [3].

Code clone detection is prominently done by performing a textual-based approach [2]. Most textual-based code clone detectors are capable of performing code clone detection without transforming the source code. NiCad

is a textual hybrid-based tool for code clone detection, that was used for this thesis [34].

2.2 Other NiCad Experiments

There have been many experiments performed using NiCad. NiCad was created in 2009 as a part of a PhD thesis project [21]. It is capable of detecting type-1, type-2, type-3, and type-4 code clones, which makes it a very capable tool for clone detection.

One of the many experiments done in NiCad was on Java. There has been an instance of clone detection in Java, that does compilation and decompilation, as a normalization step to enhance the result [35]. The compilation may be done using the standard command javac and an option for decompilation would be to use the open-source meta-programming tool procyon.

There have also been cases of detecting code clones within an Android system to find malware using NiCad [36]. It is done by finding the malware pattern within the malware code, which may lead to solutions by code clone detection. Especially, since it has been proven that there were many cloned Android applications widespread [37, 38].

3 Tools and Dataset

3.1 Dataset: BigCloneBench

BigCloneBench is a benchmark that is used to evaluate various code clone detectors in a variety of use cases [7]. It is comprised of 8 million validated clones from the big data repository IJaDataset 2.0, a repository accommodating 24,557 distinct open-source Java systems [39]. The clones are found through a heuristic search using keyword and source-code patterns. The clones are then tagged manually by judges as either true positives or false positives [13].

Table 3.1: BigCloneBench Clone Summary [13]

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4	Total
Clone Pairs	47146	4609	4191	9516	38894	5818503	8375313

There exists a BigCloneBench Clone Summary table [13]. In this table, it describes the various clone types and the amount of each clone type within BigCloneBench based on its similarity rate. Here is a list of each of the clone types with their respective acronyms and similarity rates [7, 19]:

- T1: Type-1 100%
- T2: Type-2 100%, without regarding differences in identifier names and values
- VST3: Very Strongly Type-3 90% inclusive - 100% exclusive
- ST3: Strongly Type-3 70% inclusive - 90% exclusive
- MT3: Moderately Type-3 50% inclusive - 70% exclusive
- WT3/T4: Weakly Type-3, Type-4 0% inclusive - 50% exclusive

3.2 BigCloneEval

Table 3.2: BigCloneEval Command List [13]

Command	Description
registerTool	Registers a tool with the framework.
listTools	Lists the tool(s) registered with the framework.
deleteTool	Removes a tool, and its detected clones, from the framework.
partitionInput	Partitions a clone detection input given a maximum input size.
detectClones	Automates the execution of a tool for IJaDataset.
importClones	Imports a tool's detected clones into the framework.
clearClones	Removes the imported clones of a tool from the framework.
evaluateTool	Measures the recall of a tool and produces the tool evaluation report.

BigCloneEval is a framework used to evaluate the clone detector tool using the BigCloneBench [13]. It also has a plug-in architecture, that allows the user to utilize different code clone detectors in an accessible manner. The framework produces a report consisting of different clone types that are segregated by different syntactical similarity percentages. The clones are also

separated between inter-project clones and intra-project clones. It comprises 4 major components, the referencing BigCloneBench database in the form of an H2 database, the IJaDataset repository that contains BigCloneBench, a tools database that keeps track of the tools, and the set of commands that can be used to operate BigCloneEval.

3.3 Stubber

The original BigCloneBench files could not be compiled easily due to dependencies and other factors. A tool called Stubber is used to counteract this problem. Stubber filters out the non-compilable Java files in BigCloneBench. Stubber is capable of generating over 95% of the 55,499 Java source files of BigCloneBench that are compilable [40]. It is done in order to remove the source code clone files that are not compilable so that the recall measurement is not falsified. With Stubber also comes the annotation for each of the non-compiler generated functions the metadata "startline" and "endline", which can be used as part of the input within BigCloneEval. Stubber also comes with the corresponding database change that can be used to replace the original BigCloneBench H2 database within the BigCloneEval project.

3.4 TXL

The parsing language used within NiCad is named Turing eXtender Language, also known as TXL [41]. It is a first-order functional programming language and its first usage was for rapid prototyping the language Turing [42]. It is based on Lisp [43], which uses the following features:

- Lisp list structure is used as the basis for its parse trees, grammars, and patterns
- The main control structure is based on Lisp function composition
- The functional programming aspect with complete backtracking capability for both the parser and transformer of the language

In NiCad, TXL is used to parse the input language, and also apply various transformations on it [44].

```
1   define Program
2       [Component A] [Component B] [Component C]
3   end define
4
5   rule replaceComponentA
6       replace $ [Component A]
7           _ [Component A]
8       by
9           'a
10  end rule
```

Figure 3.1: A simple example of a transformation in TXL

3.5 NiCad

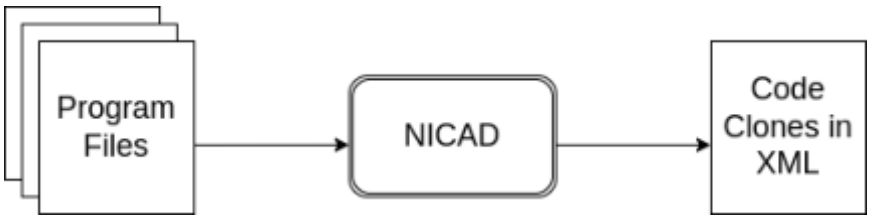

Figure 3.2: High-level view on NiCad near-miss clone detector

Automatic Clone Detection of Near-Miss Intentional Clones, also known as NiCad, is a method that hybridizes language-sensitive parsing with language-independent similarity analysis [1]. It is a code clone detection tool that has been used extensively for many clone detection research cases [35, 45, 46, 47]. It is very efficient in terms of resource usage, capable of handling even a system with more than 60 million lines by only utilizing 2 GB of memory on a standard single-processor laptop [1].

The NiCad method involves three main stages, parsing, normalization, and comparison. The parsing and normalization are done using TXL. The final stage, comparison, uses an optimized Longest Common Subsequence algorithm. FreeTXL 10.8 or later version is required to run NiCad.

The parsing and normalization are compartmentalized into plugins. Each plugin is written in TXL and will follow a centralized grammar file for the

particular language used in the analysis. In Fig. 3.2 we can see an example of how it is represented in the NiCad Clone Detector.

Various clone detection configurations are available on NiCad. This allows us to choose what kind of clones we want to have as our output. Type-1 in NiCad is classified as exact clones. On the other hand, type-2 are classified as renamed clones. Consequentially, type-3 are considered near-miss clones. Lastly, type-4 are considered semantic clones There are also subconfigurations, such as type-2c clones which are clones that are renamed consistently. There are also type-3-2 clones and type-3-3 clones, which are type-3 clones that are respectively renamed and renamed consistently. For the purpose of this thesis, we will focus on type-3 and type-4 clones.

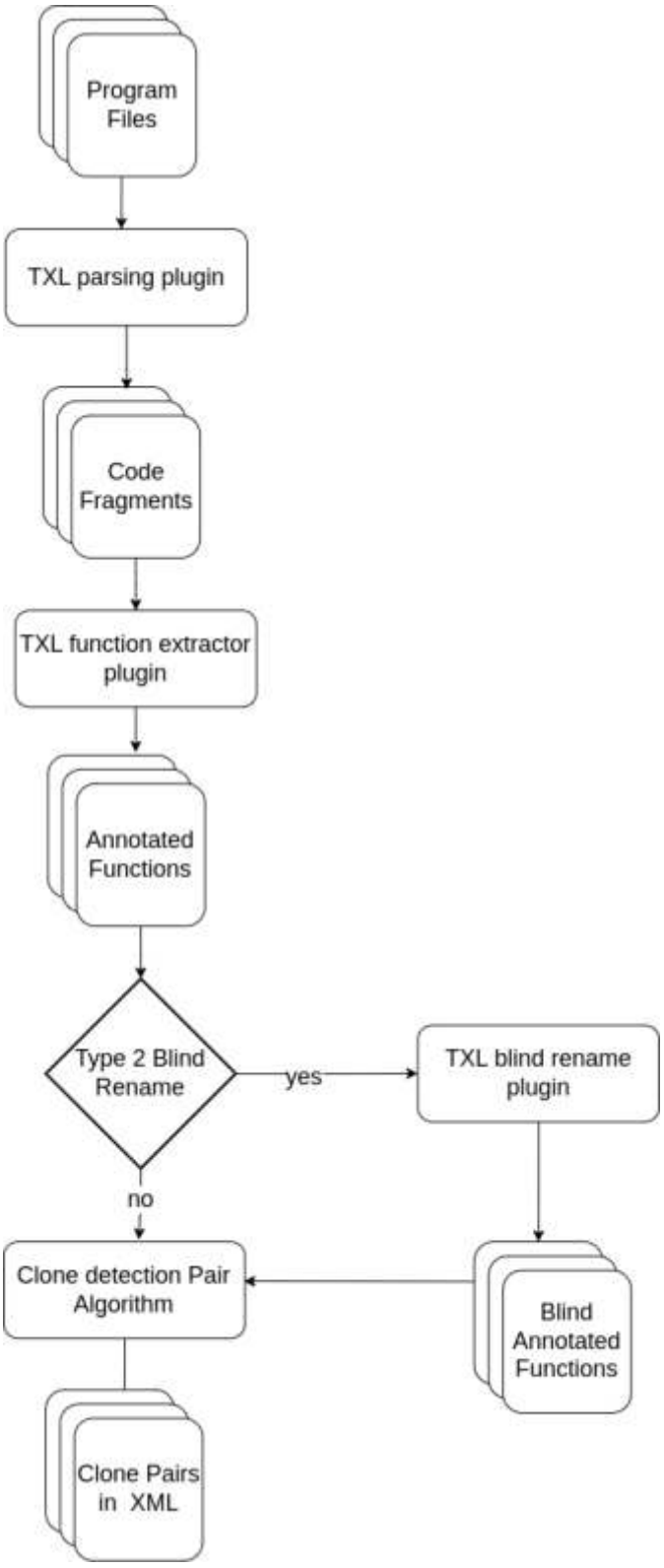

Figure 3.3: NiCad workflow

4 Processing and Integration in NiCad

4.1 Preprocessing the Dataset

Various steps are done in order to get the result for this thesis. Firstly, the BigCloneBench is stubbed and annotated using Stubber. Here are the following steps that were executed on the original java source files in Stubber [40]:

```
RuntimeInvisibleAnnotations:
    _3566.StubClass.AnnotationInterface(
      SubFolder="default"
      FileName="3566.java"
      StartLine=81
      EndLine=101
    )
```

Figure 4.1: An example of the annotations created by Stubber [40]

- JAR archives are created for each of the Java files within BigCloneBench.

- the original imports are all replaced with the StubClass

- the methods are also given an annotation as shown in figure 4.1, containing the metadata that are used in BigCloneEval. Such as the startline, endline, and original Java filename of the function. Other annotations are removed since it is not needed in clone detection

- return values are anonymized into *java.lang.Object*, other than void and variable names are renamed to prevent problems during compilation due to overloading

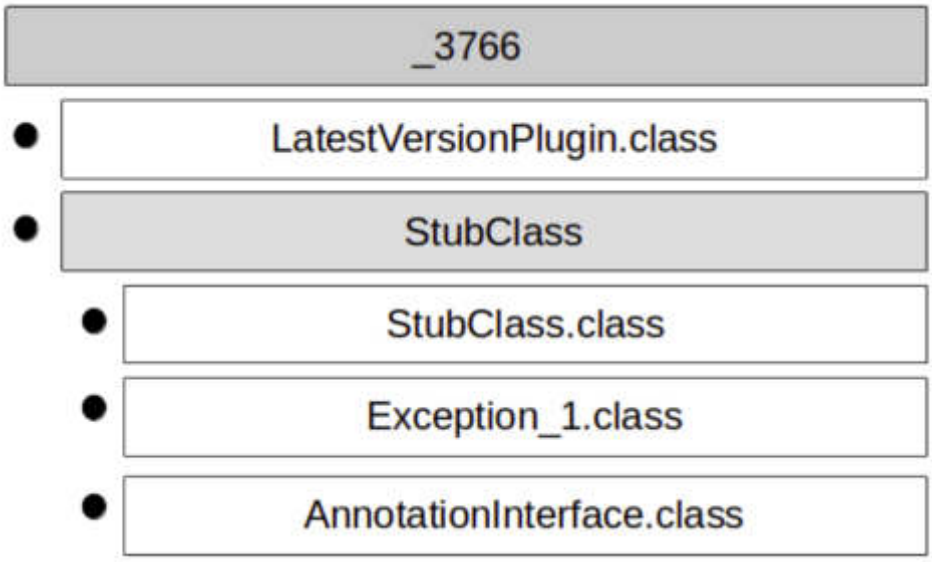

Figure 4.2: The folder structure of the JAR folder built by Stubber [40]

In the figure 4.2, we can see an example of a JAR folder created through Stubber [40]. The original Java file was 3766.java, it was then stubbed. From it, a JAR folder with the name _3766 was created. There is a compiled class in the figure with the name $LatestVersionPlugin.class$. The class filename is determined by the name of the class within the 3766.java file.

A subfolder StubClass was created such that all dependencies could become irrelevant in the compilation of the class file. Dependencies were useless in the particular process of Stubber for code clone detection [40].

The stubbed BigCloneBench was then extracted, and all of the class files were disassembled afterward to create a mapping between startline, endline, and filename in Java with their Java Bytecode equivalent. The extraction was done by moving all of the class files except the ones in the StubClass subfolder, outside of the subfolder, renaming it in this format $[Javafilename]\_[classfilename].bc$.

A mapper was needed between the original Java metadata and the metadata produced from using NiCad on the bytecode. The main reason why this required to be done was because of how BigCloneEval functions. BigCloneEval was originally used to evaluate Java code detection tools on the BigCloneBench. Due to the BigCloneBench being transformed into its JVM Bytecode intermediate representation equivalent, the metadata that are pro-

duced are then inherently changed based on the startline, endline, filename, and folder name of each of the equivalent transformed functions. To counteract this, we created the mapper and used it before the evaluation phase of our experiment.

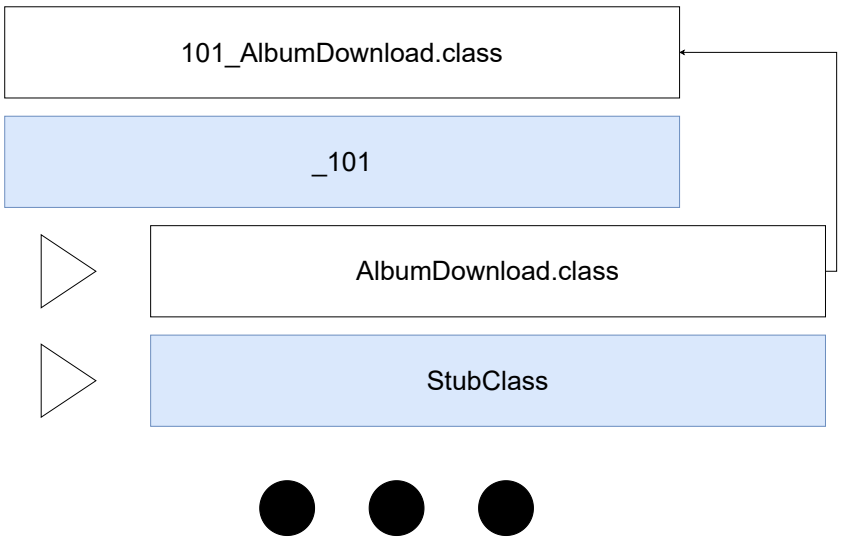

Figure 4.3: Renaming the class files from Stubber

Through this renamed BigCloneBench class file, an extractor was created to map between the original source code file metadata and the modified bytecode file metadata that will be produced at the end. In figure 4.3 we can see that by appending the java name on its prefix, we are able to properly map the class files and the java files. The main reason why this was important was because there were duplicate class file names from different Java

files. If for example hypothetically another file named "102.java" created a JAR folder from Stubber "_102", it can be possible another class file with the same "AlbumDownloader.class" exists. In this case, one of the AlbumDownloader.class will overwrite the other ones and only one Java file will be mapped to AlbumDownloader.class. The prefix in the name helps with mapping to which original Java file the class file belongs to.

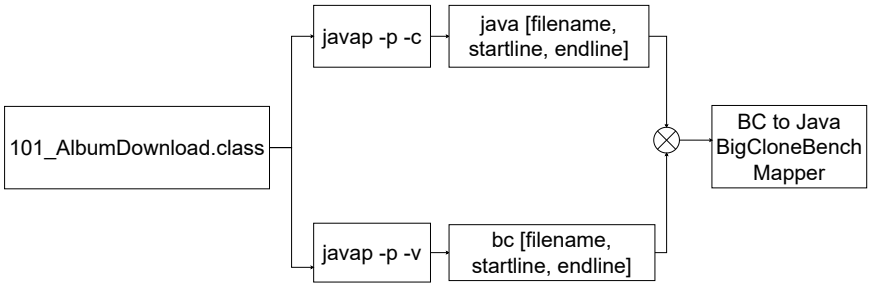

Figure 4.4: Example of making the mapper

In the first step, the renamed BigCloneBench class files were used as the input to the extractor. It has an annotation for the metadata needed for the input to BigCloneEval. The input data needed were the file name, folder name, startline, and endline of the function. The mapper contains both its original Java data and the bytecode data. The extractor retrieves that data by using `javap -p -c` and `javap -p -v` as shown in figure 4.4.

There was an important reason why the command javap was run twice. In `javap -p -v` it was possible to see the annotation that was produced

by Stubber as seen in figure 4.1, there we can see the metadata such as the subfolder, source filename, source startline, and source endline. However, the clone detection will be done on the files created from `javap -p -c`, creating the changed bytecode files metadata, consisting of the bytecode filename, bytecode startline, and bytecode endline.

```
Processing BigCloneBench_Unpacked_and_Renamed/101
    _AlbumDownloader.class
Function 0 : constructor default,101.java,18,33
Function 1 : default,101.java,35,47 has less than 15 lines.
Function 2 : default,101.java,49,80
Function 3 : default,101.java,82,97
Function 4 : default,101.java,99,102 has less than 15 lines.
Function 5 : default,101.java,104,108 has less than 15 lines.
functions to skip:  [1, 4, 5]
java filename:  101
bc filename:  AlbumDownloader
```

Figure 4.5: Log created from the extractor during the javap -p -v phase for 101_AlbumDownload.class

The extractor first ran `javap -p -v` to get the metadata needed for BigCloneEval and create a filter list based on the source code's length and the function type. In this case, we want to only use source code functions that are longer than 15 lines. We also skip every static function, static block, and JVM-generated constructors. The JVM-generated constructors are ignored since they do not exist in the source code. The filter list will keep track of the functions using a counter. The metadata for valid functions from here are extracted, which are the original source code Java filename, Java startline, and Java endline.

An example of a log from this phase can be seen in figure 4.5. There it can be seen what file was currently being processed. Each of the functions are also shown with their respective metadata. Function 0 was special, such that it is a constructor, furthermore having additional conditions to pass. It checks if the Stubber annotation is there in the constructor, if not then it is recognized as a JVM-generated constructor and is added to the functions to skip list. A more general rule is that if a function has less than 15 lines, it will be added to the functions to skip list too.

The second part of the execution of the extractor, the command `javap -p -c` was run. It then proceeds to check for each function, incrementing the counter from zero until the last function. The extractor skips the metadata extraction of the function if the function counter points to a skipped function. From here, the line of codes is tracked and used to create the bytecode metadata in the form of bc filename, bc startline, and bc endline. The end result is that the metadata from the javap -p -v and the one from javap -p -c are concatenated and creating a CSV by collecting them in this format: [java filename, java startline, java endline, bc filename, bc startline, and bc endline]

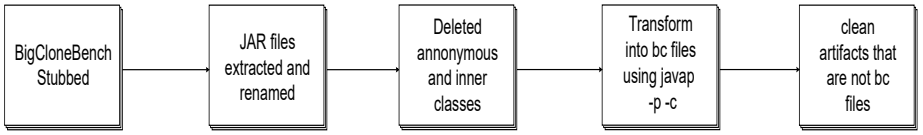

Figure 4.6: Workflow of preprocessing the Big Clone Bench

In figure 4.6, various transformations were done to the BigCloneBench that was Stubbed. It was done using multiple Python scripts, working sequentially. The first step was that the JAR files were extracted and renamed as seen in figure 4.3

The extracted BigCloneBench from the jar folders was then processed again to create Bytecode files while still retaining the folder structure. This was useful because the nicadRunner in BigCloneEval when executed using the command *detectClones* runs through each of the subfolders sequentially and appends the results of each subfolder into an output CSV.

The anonymous and inner classes were deleted to make the amount of processed data smaller. NiCad was relatively time-intensive to run and deleting it would help in getting the result for the thesis faster during the time constraint of the thesis.

Bytecode files are text files created using the command `javap -p -c`

`[target file]`. All of the artifacts, such as the StubClasses and the Java files were deleted to save space since they were not needed.

The artifacts that were in the BigCloneBench were all deleted. This includes the folder StubClass, the original source code files, the compiled .class files, and the JAR folders. In the end, there exists only the .bc version of the BigCloneBench while retaining its folder structure.

The .bc version of the BigCloneBench is then also unwrapped for easier parsing using a bash script within Bytecode Texelite, such that there are no shortcuts within the JVM bytecode class files being used, as an example iload_1 will instead be iload 1.

4.2 Extending NiCad

Extending NiCad is crucial for it to be able to process bytecode files and register code files as input. Preparations have to be done beforehand to make it work. First NiCad 6.2 was downloaded from the website www.txl.ca. Afterward, TXL v10.8b and Turing+ compiler version 6.2 were downloaded and installed from the same website.

NiCad was directly extended within the repository itself using the program-

ming language TXL. The following files were created in preparation for the experiment:

- bc.grm
- bc.txl
- bc-extract-functions.txl
- bc-rename-blind-functions.txl

```
% TXL BC Grammar
include "bc.grm"

% Ignore BOM headers
include "bom.grm"

% Just parse
function main
    replace [program]
        P [program]
    by
    P
end function
```

Figure 4.7: File named bc.txl, developed in NiCad for this thesis

There were various functions that these TXL files accomplished. Firstly, bc.txl was used as a compiler directive to show that the ending bc is a valid option within NiCad.

In bc.grm a parser was built within it using TXL. Every situation within a bytecode file had to be accounted for to make a working parser. It is a top-down parser that starts with the whole program. It goes from the program

to the class files, and then it will resolve the file header created by javap first and then the class itself. The class is defined by its header and body. Within the body, there exist various functions and other things such as static variables, static blocks, and an exception table. First static variables are resolved. The static variables are ignored during the extraction phase. After the static variables are the normal functions. Here within it exist various JVM operands with different rules. The rules are defined explicitly within the grammar file. Optionally exist the exception table within the function body. After the functions come optionally the static blocks.

Ensuing the parser, exist the bc-extract-functions.txl. It extracts a part of the parsed Abstract Syntax Tree and uses it as clone fragments. The startline and endline are then accounted for and will be printed out in a resulting XML file.

To conclude, bc-rename-blind-functions.txl was used to replace each JVM operand within the functions to be changed into a replacement that removes its ambiguity due to different numbers, setting all of its values uniformly to 0. All of the other IDs are replaced by x if it was not a part of the JVM instructions, such as return value or function names. This step is especially important, since in the bytecode, the JVM operands are often very different, resulting in a very different similarity rate. Without this step, type-1 and type-2 clones could barely be found.

After the changes to the txl/ folder, it was required to run *make* in the root directory to finalize the changes that were made. This was also an important step to be done during debugging, as it would also show the compilation errors if they were to exist.

```
cd path-to-NiCad
./nicad6 functions bc "$path" type3-2-custom > /dev/null 2> /
    dev/null
# Convert detected clones into the format that BigCloneEval
    uses
cat ${path}_functions-blind-clones/${dir}_functions-blind-
    clones-{threshold_number}.xml | sed 's$<source file="$$g' |
    sed 's$" startline="$,$g' | sed 's$" endline="$,$g' | sed '
    s$" pcid=.*"></source>$$g' | sed 's$<clone nlines=.*$$g' |
    sed 's$</clone>.*$$g' | sed 's$</clones>$$g' |sed 's$<clones
    >$$g' | sed 's$<cloneinfo.*$$g' | sed 's$<systeminfo.*$$g' |
     sed 's$<runinfo.*$$g' | sed '/^$/d' | paste -d ',' - - |
    sed "s#${path}/##g" | sed 's#/#,#g'
```

Figure 4.8: The nicadRunner script in BigCloneEval

To integrate NiCad with BigCloneEval, a utility file called nicadRunner was used. It is located in the subfolder "sample" within BigCloneEval. This are the steps that were executed in the nicadRunner:

- change directory to NiCad
- run NiCad while suppressing its output in the stdout
- open the resulting clone XML files and filter out all of the XML component, transforming it into a CSV based on the BigCloneEval input format

The mapper creates a mapping between the Java source code metadata with the JVM bytecode metadata of BigCloneBench. The BigCloneBench was transformed into its JVM bytecode equivalent while retaining its folder structure. Coincidentally, Nicad was extended to be able to detect clones on the JVM bytecode. And finally, the nicadRunner in BigCloneEval was updated to accommodate the experiments. With all of the preparation done on the BigCloneBench dataset, the mapper, the NiCad extension, and the nicadRunner, it was now possible to run the experiment.

5 Experiments

This chapter explained how the experiments were set up and the results of the experimentations that were done during the writing of the thesis. The results were also compared with StoneDetector [48] and IClones [49] for bytecode clone detections.

5.1 Setting up the Measurement

Many software tools were used in conjunction with this thesis to answer the research questions. The following software tools and hardware were employed:

- Software Tools
  - BigCloneEval Version 0.1

- BigCloneBench Version 1.0 (2016-06-19) from Stubber
- Stubber
- H2 Database from Stubber to replace the one originally in BigCloneEval
- Bytecode Texelite Version 1.0
- TXL v10.8b
- NiCad version 6.2 (13.11.20) extended with plugins for JVM bytecode and Jimple registry code
- openjdk 21.0.3 2024-04-16
- Python 3.12.3 used as a scripting language within Bytecode Texelite
- GNU bash, version 5.2.21(1)-release (x86-64-pc-linux-gnu) used for additional scripting within Bytecode Texelite
- Ubuntu version 24.04, utilizing the UNIX filesystem to make NiCad work
- Linux 6.8.0-40-generic kernel version

  - Turing+ Compiler version 6.2 for compiling the Turing files within NiCad

  - Miller version 6.11.0 for processing the CSV files

- Hardware

  - 12th Gen Intel® Core™ i7-12650H x 16 processor

  - 16.0 GiB Memory

  - Nvidia GeForce RTX™ 4070 Laptop GPU

5.2 Obtaining Recall Results in BigCloneEval

In this thesis, we are interested in finding out the recall obtained from BigCloneEval. Recall is the ratio of the clones detected, compared to the clones that exist within the system [50].

It is done through various commands within BigCloneEval. Foremost, we need to register NiCad as a tool in BigCloneEval using the command `registerTool`. `detectClones` was used in conjunction with NiCad to find the code clones in each of the subfolders of BigCloneBench. This command has produced an output.csv file, containing the code clones, with the meta-

data of the following: parent folder, filename, startline, endline. These metadata have to be mapped back to their original Java form since BigCloneEval was created to evaluate Java clone detectors for BigCloneBench.

```
threshold=0.25, 0.22, or 0.18
minsize=10
maxsize=2500
transform=none
rename=blind
filter=none
abstract=none
normalize=none
cluster=yes
report=no
```

Figure 5.1: Type-3-2 Config in NiCad

Type 3-2 configuration in NiCad was used, such that blind renaming would have been applied to the extracted functions in NiCad by using the rename=blind option in the config file. The config file can be seen in figure 5.1. This is especially useful for JVM bytecode because the JVM operand differs in each line within a function in the JVM bytecode. The similarity thresholds for this thesis that were used are the following: 0.25, 0.22, and 0.18. No pre-transformations were used so the option transform was set to none. No normalization, filter, or additional abstractions were made to the Abstract Syntax Tree intermediate representation. The cluster is set to yes, such that

the clustering algorithm could work on the resulting Longest Common Subsequence. The report is set to no because in BigCloneEval the command *detectClones* takes the result of the nicadRunner in the *stdout* using the *cat* and piped through several filtering *sed* Unix commands, as seen in the figure 4.8 from the previous chapter.

Subfolder number 4 in BigCloneBench is the biggest subfolder in the project. With the JVM Bytecode of stubbed BigCloneBench as the input, it produces too many clones and it cannot be run directly in NiCad. To counteract this problem, it is possible to use a partitioning script within BigCloneEval. This will result in a massive output CSV file. To solve this problem, the CSV file was split using a CLI tool named Miller. The split CSV files were then mapped into a single output file and then appended to the rest of the clones.

To obtain the result report, we have to use the command `importClones` on the mapped output file, and then `evaluateTool`. This would produce the report file, stating the recall rate of the clones that were detected.

Thresholds	25%		22%		18%	
Clone Type	Amount	Recall	Amount	Recall	Amount	Recall
T-1	20756	99.82%	20756	99.82%	20756	99.82%
T-2	3460	99.71%	3460	99.71%	3460	99.71%
VST-3	3170	97.68%	3169	97.66%	3166	97.57%
ST-3	6892	94.42%	5885	80.63%	5160	70.69%
MT-3	8369	32.81%	2884	11.30%	1327	5.2%
WT-3/T-4	14337	0.35%	1350	0.03%	317	0.01%
Total Clones Found	655,862		645,483		548,829	

Table 5.1: Evaluation result for bytecode in BigCloneEval using preprocessed BigCloneBench

In table 5.1 we can see the result of the experiments performed on the BigCloneBench using NiCad and evaluated using BigCloneEval. The number of clones pairs detected is based on the clones that were manually found and marked in the H2 Database [13] and then filtered out in Stubber, due to Stubber using a subset of the BigCloneBench dataset [40]. The table shows the results procured by testing the different threshold configurations (0.25, 0.22, 0.18) on the stubbed and transformed into its JVM Bytecode intermediate representation BigCloneBench. The recall rate in the table shows how many of the detected clone pairs out of all the marked clone pairs.

Throughout type-1, type-2, and the different thresholds, there are no differ-

ences in the amount of clones detected. There are 20756 type-1 clones with a recall rate of 99.82% in all three thresholds. General type-2 clones that were detected were 3460 clones with a 99.71% recall rate.

For the very strongly type-3 clones, the results obtained from running NiCad on the bytecode version of the BigCloneBench through evaluating it in BigCloneEval are relatively small. The very strongly type-3 clones were found with the amount of 3170, 3169, and 3166 in the threshold respectively 25%, 22%, and 18%. The recall rates for the very strongly type-3 clones are each 97.68%, 97.66%, and 97.57% for each threshold configuration.

However, the strongly type-3 clones experienced a moderate drop from each threshold value. On threshold configuration of 25% 6893 clones were detected with 94.42% recall rate, while on 22% 5885 clones were found with 80.63% recall rate and lastly on 18% 5160 clones were located with 70.69% recall rate. Between threshold 25% and 22% there was a 13.79% recall rate drop and between threshold 22% and 18% the recall rate was dropped by 9.94%.

Despite that, the moderately type-3 clones experienced a stronger drop between the thresholds. On threshold 25%, it starts with 8369 clones discovered with a recall rate of 32.81%. Afterward, on threshold 22% were 2884 clones identified with the recall rate of 11.30%. Finally, on threshold 18% 1327 clones were recognized with a recall rate of 5.2%. The drop in the recall rate

of the moderately type-3 clones based on different threshold configurations was much sharper than the strongly type-3 clones. From 25% to 22% threshold the recall rate was almost 3 times less, while from 22% to 18% was 2 times less.

Finally, the weakly type-3 and type-4 clones are put in the same category in BigCloneEval. The number of clones and recall rate found were 14337 clones with 0.35% recall rate, 1350 clones with 0.03%, and 317 clones with 0.01% recall rate for each of the thresholds 25%, 22%, and 18% in the configuration. Here we can see a very sharp decline in the recall rate. From threshold 25% to 22% the recall rate is less than 10x, and between threshold 22% to 18%, the recall rate is 3x less.

The total number of clones was as expected lower after the thresholds were decreased. This was due to NiCad using the similarity threshold when comparing the clones, to see how similar two extracted clone fragments have to be to be classified as a clone pair. At the 25% threshold, we were able to find 655,862 clones, at the 22% threshold, 645,483 clones were found, and finally, at the 18% threshold, 548,829 clones were found in this threshold.

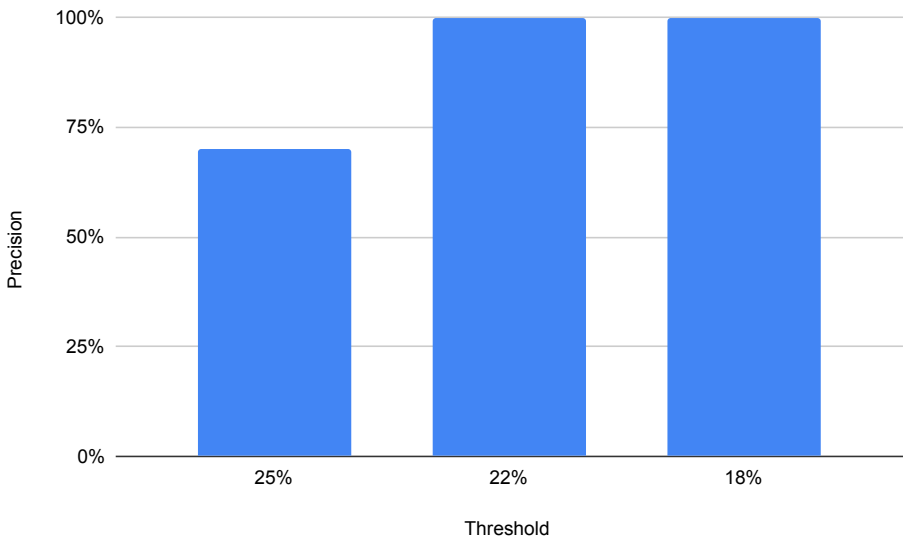

Figure 5.2: NiCad precision based on its threshold on bytecode

The precision of the clone detector NiCad was also assessed as seen in figure 5.2. From each mapped output produced from the command *detectClones* on each of the threshold configurations, 100 clone pairs were randomly picked. The functions from the clone pairs are then manually looked for in the source files. The picked function pairs were then checked manually by the author and tagged whether it was a real clone pair or not. Real clone pairs based on the clone definition explained in the first chapter of this thesis. The precision was then calculated by seeing how many were deemed real clone pairs out of the clone pairs obtained from the supervisor.

In figure 5.2, It can be seen that at threshold 25% only 70 of the 100 clone pairs were deemed as real clone pairs, scoring a 70% in the precision. The

precision at threshold 22% and 18% are both at 100%, as the function pairs that were sent are all real clone pairs.

	NiCad 18% (BC)		StoneDetector		iClones	
Clone Type	Amount	Recall	Amount	Recall	Amount	Recall
T-1	20756	99.82%	20831	99.9%	20842	99.9%
T-2	3460	99.71%	3476	100%	3453	99.3%
VST-3	3166	97.57%	3272	99.4%	3253	98.9%
ST-3	5160	70.69%	6702	90.5%	6531	88.2%
MT-3	1327	5.2%	4335	16.1%	4051	15.1%
WT-3/T-4	317	0.01%	2691	0.03%	6554	0.1%
Total Clones Found	548,829		621,169		987.659	
Precision	100%		99%		97%	

Table 5.2: Comparing result of NiCad on Bytecode at 18% threshold against StoneDetector and iClones on the source code [16]

In table 5.2, we compare the result of one of our experiments, namely the NiCad bytecode experiment with the threshold of 18% with StoneDetector [48] and iClones [49] results obtained from a previous comparison research [16]. The clones that are observed for recall rate are the clones that were previously marked in the BigCloneBench dataset, with the special case of NiCad in this table by choosing a slightly smaller subset due to using Stubber. In this case, we can see that the number of clones detected for type-1, type-2, and very strongly type-3 are very close to each other. Type-1

and type-2 clones differ by less than 1%. With NiCad using the threshold config at 18% for bytecode, the strongly type-3 clones detected were only at 70.69%, compared to StoneDetector at 90.5% and iClones at 88.2%. At moderately type-3 clones, NiCad suffers heavily in terms of clones detected. NiCad with this configuration could only detect 5.2% of the marked moderately type-3 clone, while StoneDetector could find 16.1% and iClones 15.1%, which in numbers translates to 4335 and 4051 clones respectively, and for NiCad only 1327 clones. On weakly type-3 and type-4 clones, all three clone detectors once again experience a massive drop. NiCad with that configuration could find 317 weakly type-3 and type-4 clones, while StoneDetector only had 2691 clones and iClones had 6554 clones. The total number of unique clones found using NiCad at 18% threshold for Bytecode was 548,829 clones, in StoneDetector was 621,169 clones and finally in iClones 987,659 clones. The precision rate between all three clone detectors was relatively the same at almost 100%

6 Discussion

It can be seen that detecting code clones using an intermediate representation enables it to find semantic clones. Semantic clones are very hard to detect due to their nature of not having the same syntactical structure. There have been cases where semantic clones were found between dotNet languages [20].

In this thesis, Java files from stubbed BigCloneBench were transformed into its Java bytecode equivalent, and an extended NiCad was executed on it. There was a stark difference that could be seen between the non-existent type 4 clones detected in Java and the numerous ones found using the Java bytecode intermediate representation.

It also shows that using different threshold configurations may differ the precision of the clone detection. It is highly likely that the lower the threshold configuration in NiCad, the more likely it is to be more precise. The trade-off will however be that the number of clones that are detected with a lower

threshold configuration will be a lot less compared to a higher threshold configuration.

The recall rate of the clones could be better if the anonymous and inner classes were also included. However, due to the time it takes for it to load would be much longer, the author has decided to omit the anonymous and inner classes in the analysis to be able to produce adequate results within the time constraint of the thesis.

The precision could be improved further by marking the whole database in a future study. By doing so, it enables other researchers to evaluate the precision of their detection tool easier. The current method used a randomized pool of 100 function pairs for each threshold configuration that were marked manually. It could be possible that the precision would be much higher or lower than its true precision, simply due to the luck-based randomness of the picked function pairs.

The author has also decided to focus more on the JVM bytecode intermediate representation of the Java file, instead of analyzing the Jimple source code intermediate representation too. This decision was made due to the interest in the results produced from the JVM bytecode intermediate representation and to go deeper within it, pursuing more insights into clone detection using JVM bytecode. Another reason was due to the time constraint of the thesis

itself. In the future, it would be an interesting topic to explore for future researchers to expand upon.

The very low recall rate of semantic clones could also be a possible research field open for future exploration. The author believes there should be another clever way of detecting clones that could potentially detect semantic clones without sacrificing much of the precision rate.

7 Conclusion

Based on the results of the experiments and the discussion regarding code clone detection on JVM Bytecode, it has been concluded that using it enables us to systematically detect source clones. The recall rate was really good on type-1 and type-2 clones. It answered the research question of whether it is possible to use JVM bytecode to detect source code, concluding that it is indeed possible.

The semantic clones (Weakly Type-3 and Type-4) would have been undetectable by NiCad if it was run directly on their respective programming language instead. From the evaluation results, we can see that plenty of semantic clones were detected at higher threshold configurations in NiCad when using a JVM bytecode intermediate representation of BigCloneBench on BigCloneEval. However, the result for type-3 and type-4 clones was still inferior to StoneDetector and IClones.

Bibliography

[1] James R. Cordy and Chanchal K. Roy. "The NiCad Clone Detector". In: *2011 IEEE 19th International Conference on Program Comprehension*. 2011, pp. 219–220. DOI: 10.1109/ICPC.2011.26.

[2] Qurat Ul Ain et al. "A Systematic Review on Code Clone Detection". In: *IEEE Access* 7 (2019), pp. 86121–86144. DOI: 10.1109/ACCESS.2019.2918202.

[3] Giuseppe A Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. "An approach to identify duplicated web pages". In: *Proceedings 26th Annual International Computer Software and Applications*. IEEE. 2002, pp. 481–486.

[4] Chanchal K Roy, James R Cordy, and Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". In: *Science of computer programming* 74.7 (2009), pp. 470–495.

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

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

[7] Jeffrey Svajlenko and Chanchal K Roy. “Evaluating clone detection tools with bigclonebench”. In: *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2015, pp. 131–140.

[8] Markus Dahm. “Byte code engineering”. In: *JIT’99: Java-Informations-Tage 1999*. Springer. 1999, pp. 267–277.

[9] Raja Vallée-Rai et al. “Soot: A Java bytecode optimization framework”. In: *CASCON First Decade High Impact Papers*. 2010, pp. 214–224.

[10] Andre Schafer, Wolfram Amme, and Thomas S. Heinze. “Stubber: Compiling source code into bytecode without dependencies for java code clone detection”. In: *2021 IEEE 15th International Workshop on Software Clones (IWSC)* (Oct. 2021). DOI: 10.1109/iwsc53727.2021.00011.

[11] Jozef Kostelansky and Lubomir Dedera. "An evaluation of output from current Java Bytecode decompilers: Is it android which is responsible for such quality boost?" In: *2017 Communication and Information Technologies (KIT)* (Oct. 2017). DOI: 10.23919/kit.2017.8109451.

[12] Chanchal K Roy and James R Cordy. "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization". In: *2008 16th iEEE international conference on program comprehension*. IEEE. 2008, pp. 172–181.

[13] Jeffrey Svajlenko and Chanchal K Roy. "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench". In: *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2016, pp. 596–600.

[14] Noah Mauthe, Ulf Kargen, and Nahid Shahmehri. "A large-scale empirical study of Android app decompilation". In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Mar. 2021). DOI: 10.1109/saner50967.2021.00044.

[15] Kai Chen, Peng Liu, and Yingjun Zhang. "Achieving accuracy and scalability simultaneously in detecting application clones on android markets". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 175–186.

[16] André Schäfer. “Codeklonerkennung mit Dominatorinformationen”. PhD thesis. Dissertation, Jena, Friedrich-Schiller-Universität Jena, 2023, 2023.

[17] Chanchal Kumar Roy and James R Cordy. “A survey on software clone detection research”. In: *Queen's School of computing TR* 541.115 (2007), pp. 64–68.

[18] Stefan Bellon et al. “Comparison and evaluation of Clone Detection Tools”. In: *IEEE Transactions on Software Engineering* 33.9 (Sept. 2007), pp. 577–591. DOI: 10.1109/tse.2007.70725.

[19] Jeffrey Svajlenko et al. “Towards a big data curated benchmark of inter-project code clones”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 476–480.

[20] Farouq Al-Omari et al. “Detecting clones across microsoft. net programming languages”. In: *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 405–414.

[21] Chanchal K Roy. “Detection and analysis of near-miss software clones”. In: *2009 IEEE International Conference on Software Maintenance*. IEEE. 2009, pp. 447–450.

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

[23] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. “CCFinder: A multilinguistic token-based code clone detection system for large scale source code”. In: *IEEE transactions on software engineering* 28.7 (2002), pp. 654–670.

[24] Toshihiro Kamiya. “Ccfinderx: An interactive code clone analysis environment”. In: *Code Clone Analysis: Research, Tools, and Practices* (2021), pp. 31–44.

[25] Mohammad Reza Farhadi et al. “Scalable code clone search for malware analysis”. In: *Digital Investigation* 15 (2015), pp. 46–60.

[26] Ira D Baxter et al. “Clone detection using abstract syntax trees”. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE. 1998, pp. 368–377.

[27] Lingxiao Jiang et al. “Deckard: Scalable and accurate tree-based detection of code clones”. In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 96–105.

[28] Tijana Vislavski et al. “LICCA: A tool for cross-language clone detection”. In: *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2018, pp. 512–516.

[29] Gordana Rakić. “Extendable and adaptable framework for input language independent static analysis”. PhD thesis. University of Novi Sad (Serbia), 2015.

[30] Jeffrey Svajlenko and Chanchal Kumar Roy. “Fast and flexible large-scale clone detection with CloneWorks.” In: *ICSE (Companion Volume)*. 2017, pp. 27–30.

[31] Iman Keivanloo, Chanchal K Roy, and Juergen Rilling. “SeByte: Scalable clone and similarity search for bytecode”. In: *Science of Computer Programming* 95 (2014), pp. 426–444.

[32] Zhenmin Li et al. “CP-Miner: Finding copy-paste and related bugs in large-scale software code”. In: *IEEE Transactions on software Engineering* 32.3 (2006), pp. 176–192.

[33] Martin White et al. “Deep learning code fragments for code clone detection”. In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 2016, pp. 87–98.

[34] Gurpreet Singh et al. “To enhance the code clone detection algorithm by using hybrid approach for detection of code clones”. In: *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE. 2017, pp. 192–198.

[35] Chaiyong Ragkhitwetsagul and Jens Krinke. “Using compilation/decompilation to enhance clone detection”. In: *2017 IEEE 11th international workshop on software clones (IWSC)*. IEEE. 2017, pp. 1–7.

[36] Jian Chen et al. “Detecting android malware using clone detection”. In: *Journal of Computer Science and Technology* 30 (2015), pp. 942–956.

[37] Jonathan Crussell, Clint Gibler, and Hao Chen. "Attack of the clones: Detecting cloned applications on android markets". In: *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*. Springer. 2012, pp. 37–54.

[38] Wu Zhou et al. "Detecting repackaged smartphone applications in third-party android marketplaces". In: *Proceedings of the second ACM conference on Data and Application Security and Privacy*. 2012, pp. 317–326.

[39] Ambient Software Evolution Group. *SECold IJaDataset 2.0*. Jan. 2013. URL: `https://sites.google.com/site/asegsecold/projects/seclone`.

[40] André Schäfer, Wolfram Amme, and Thomas S Heinze. "STUBBER: compiling source code into bytecode without dependencies for java code clone detection". In: *2021 IEEE 15th International Workshop on Software Clones (IWSC)*. IEEE. 2021, pp. 29–35.

[41] James R. Cordy. "The TXL source transformation language". In: *Science of Computer Programming* 61.3 (2006), pp. 190–210. DOI: `10.1016/j.scico.2006.04.002`.

[42] Richard C Holt and James R Cordy. "The Turing programming language". In: *Communications of the ACM* 31.12 (1988), pp. 1410–1423.

[43] John McCarthy. “History of LISP”. In: *History of programming languages*. 1978, pp. 173–185.

[44] James R Cordy. “TXL source transformation in practice”. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. 2015, pp. 590–591.

[45] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. “Similarity of source code in the presence of pervasive modifications”. In: *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)*. IEEE. 2016, pp. 117–126.

[46] Jeffrey Svajlenko, Iman Keivanloo, and Chanchal K Roy. “Big data clone detection using classical detectors: an exploratory study”. In: *Journal of Software: Evolution and Process* 27.6 (2015), pp. 430–464.

[47] Tiantian Wang et al. “Searching for better configurations: a rigorous approach to clone evaluation”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 455–465.

[48] André Schäfer, Wolfram Amme, and Thomas S Heinze. “StoneDetector: Structural and Sub-Clone Detection”. In: *2023 IEEE 17th International Workshop on Software Clones (IWSC)*. IEEE. 2023, pp. 33–36.

[49] Nils Göde and Rainer Koschke. “Incremental clone detection”. In: *2009 13th European conference on software maintenance and reengineering*. IEEE. 2009, pp. 219–228.

[50] Jeffrey Svajlenko and Chanchal K Roy. “Evaluating modern clone detection tools”. In: *2014 IEEE international conference on software maintenance and evolution*. IEEE. 2014, pp. 321–330.