University of Magdeburg
School of Computer Science

Bachelor Thesis

Analyzing the Robustness of Clone Detection Tools Regarding Code Obfuscation

Author:

Daniel Meyer

October, 2012

Advisors:

Dipl.-Inform. Sandro Schulze

Institute for Technical and Business Information Systems

Meyer, Daniel:
Analyzing the Robustness of Clone Detection Tools Regarding Code Obfuscation
Bachelor Thesis, University of Magdeburg, 2012.

Abstract

Research has shown that 7% to 23% of a typical source code system consists of cloned code. Some clones are introduced intentionally, but a majority is unintenionally created. To find these clones, several code clone detection tools have been developed. They are used in several fields such as detection of software plagiarism, malware detection or code quality enhancing. However, this process is very computation intensive and takes a lot of time depending on the size of the source code. There's a technique called code obfuscation to further complicate the detection of code clones. The aim of code obfuscation is to obscure source code fragments (in this case code clones).
The purpose of this study is to investigate the effects of source code obfuscation on the performance of code clone detection tools. To this end, we conducted a case study to evaluate the robustness of these tools.
For the case study, we developed an obfuscator to automatically apply several obfuscations on copies of the subject system. We then analyzed the obfuscated copies in combination with the original code with four clone detectors, each using a different technique (textual, token, AST, PDG).
Our results reveal that the token- and tree-based detection technique are robust against simple transformations, but are vulnerable against more complex ones. In contrast, the PDG-based technique is more robust against complex obfuscations. Finally, the textual-based technique showed no robustness against obfuscations at all.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Reusing code fragments in software systems is a common activity during development. Usually, it is performed in an ad-hoc fashion by intentionally copy, paste and modify (if necessary), or by unintentionally writing equal code with minor differences from another fragment. Such fragments are known as code clones.
Former research has shown, that 7% to 23% of typical code is a result of cloning code fragments. With this significant amount of clones it is important to analyze the effects of code cloning [Bak95] [RC08].
Commonly, code clones are considered to have a negative effect on the overall quality and reliability of source code. For instance, software maintenance can be made more difficult - if a bug is detected in one fragment, all similar fragments have to be checked. Due to the increase of redundant code, code quality and code understanding suffers from bad design and more time is needed to understand the code. Furthermore, resource requirements increase due to the higher growth rate of the system size [RC07]. However, there are certain benefits that come from code cloning, which includes a clean software architecture [KG06].

Software plagiarism is another domain, where code clones are relevant. Instead of analyzing one single software system for code clones, it is possible to find code clones between two or more systems. If the result identifies the majority of the systems as code clones, it is possible that one of the systems is a copy of the other, or certain parts are copied.

To detect replicated code fragments, a variety of code clone detectors have been developed. Code clone detectors serve the purpose to analyze one or more systems to identify similar parts, extract them and present them in a user-readable format to the user for further actions. This is done by different detection techniques that depend on different source code representations. The majority of developed detectors are text-, token-, tree- or PDG-based [RCK09]. Each technique has its own strengths and weaknesses. Existing studies evaluated code clone detectors for their performance to find

code clones, and then rate them depending on their results [RCK09] [RC07]. However, little is known why certain detectors have better or worse results.

Goal of this Thesis

The goal of this thesis, is to evaluate the robustness of different code clone detection techniques considering obfuscated code. Here, we will use a code obfuscator to obscure an existing software system, which will be analyzed by different code clone detectors. We will analyze the results of this process, to identify the effect of certain code transformations. This will give a conclusion, if and why the detectors are robust against certain transformations.

Finally, we present and discuss our results, especially whether and why different techniques are prone to different obfuscations. Our subject system is Java project consisting of 21 source files and 2415 lines of code. The developed obfuscator is used to apply different transformations on several copies of the original project.

Structure of the Thesis

This thesis is structured as follows: In Section 2, we present the required background knowledge to further understand the thesis. There, we present the basics of code cloning while discussing the types of code clones and a typical code clone detection process. Additionally, we present four code clone detection implementations, with each one using a different technique. Code obfuscation builds the second part of the background section. We present a code obfuscation, classify different transformations and discuss the application of code obfuscation.

In Section 3, we introduce the code obfuscator *ARTIFICE* we created. There we discuss several requirements the obfuscator has to meet and give an insight of our implementation. Additionally, a quick survey of obfuscations the obfuscator can apply is given.

In Section 4, we present the conducted case study. We will present how we approached the study and discuss our set-up. Then, we discuss the results of four different code clone detection processes and analyze them, to evaluate the robustness of each technique. At last, we summarize our evaluation and evaluate its threats to validity.

Finally, we give a summary of this thesis, discuss works that are related to this work, and give an outlook for future work.

2. Background

In this section we provide the theoretical foundation for further understanding of this thesis. The first part introduces the basics of code clone detection, with a brief overview of detection techniques and important terms used in that field. In the second part, we give an introduction to the concept of code obfuscation.

2.1 Code Clone Detection

During the development/evolution phase of a software system it often occurs that the developer reuses existing code fragments by copy/pasting, thus creating code clones.

There are several reasons for a developer to clone existing code, for example: A huge part of code clones gets introduced because of a time limit that is assigned to the developer. To spare time, he uses the easy way and looks for similar existing solutions and adapts it to the current problem. Sometimes code clones are intentionally created for the benefit of a clean and understandable software architecture [KG06].

While some code clones are intentionally created and beneficial, there are several drawbacks that come with code cloning. Frequent code cloning results in a high growth rate of the system, thus increasing the resource requirements. That may create a problem for compact devices or other devices with limited resources. Maintenance costs rise with every code clone, especially if a bug is found in the original fragment. Not only does the original fragment require bugfixing, every fragment that derived from it needs to be handled with. Extensive cloning may introduce bad design, making it difficult to understand and reuse the part of the system [Joh94] [AM02] [JM96].

If the code clones in a software system show a detrimental effect on the code quality, a code clone detector can be used to detect potential code clones. Found clones can then be handled with.

There are other tasks where clone detection grants a huge benefit. Some of these are:

- **Detection of malicious software:** Different malicious software families may contain parts where they are similar. Clone detection can be used to find similarities between two families, and find evidence of its malicious intends [WL06].
- **Detection of plagiarism and copyright infringements:** Similar code between to software systems can signalize a copyright infringement if the original code is not intended for free use. Clone Detectors can find these infringements [Bak95].
- **Code compacting:** Source code size can be reduced if the clones get eliminated [WKCG03].

2.1.1 Code Clone Types

Two fragments can be similar in two different ways: they can have a similarity in their source code or they are similar in their functionalities without being textually similar. Each domain contains different code clone types where a code clone can be assigned to. The types are the following:

- Textual Similarity:

  Type I: Identical code fragments except for variations in whitespace, layout and comments.

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

  Type III: Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

- Functional Similarity:

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

2.1.1.1 Type I

A *Type I* clone is a code clone fragment which is identical to the original fragment. Only whitespace(new lines, blanks, ...), comments and layout changes are allowed to differ between both fragments.

```
1 int b = 20;
2 int c = 0;
3 for {int a = 0; a < b; a++} {
4     b = a + b;  // comment
5     c = b;          // another comment
6 }
```

Listing 2.1: Original Code

A *Type I* clone of the original fragment could be as follows:

```
1 int b=20;
2 int c=0;
3 for{int a=0;a<b;a++}{ //comment
4     b=a+b;
5     c=b;} //another comment
```

Listing 2.2: Type I - Code Clone

These two fragment are essentially the same after removing blanks and reordering the brackets. Even with the changed layout and the different comments, it still would be a clone. A capable clone detector is likely to find each *Type I* clone because of its simplicity. Only line-by-line detectors would have problems, because the location of brackets can create different lines.

2.1.1.2 Type II

A *Type II* clone inherits all properties of a *Type I* clone. In Addition, all user-defined identifiers can be different from the original. Let's consider listing 2.1 as the original again. A *Type II* clone could be as follows:

```
1 int d=10;
2 int e=5;
3 for {int i =0;i<e;i++}
4 {
5     d = i + d;  // comment
6     e = d; } // another comment
```

Listing 2.3: Type II - Code Clone

Compared to the original fragment some changes in the layout and whitespace were introduced. Furthermore, all identifiers have been changed but the syntactic structure remains the same.

2.1.1.3 Type III

A *Type III* clone is a copied fragment that has all characteristics of a *Type II* clone, with the addition that statements can be added, deleted or changed. With listing 2.1 as original, a *Type III* clone could be:

```
1 int d=10;
2 int e=5;
3 for {int i =0;i<e;i++}
4 {
5     d = i + d;  // comment
6     d = d;           // new statement
7     e = d;
8 } // another comment
```

Listing 2.4: Type III - Code Clone

With the addition of the new statement, the fragment becomes a *Type III* clone.

2.1.1.4 Type IV

In contrast to the other types, *Type IV* clones do not rely on textual similarities. Rather, two code fragments have to be similar in a semantic way, which means that the clone is not necessarily a copy of the original. A behavioral clone exists, if two fragments have the same pre-/postconditions. An example of a *Type IV* clone would be the following:

```
1 int i = 1;
2 int j = 1;
3 for (i=1; i<=f; i++)
4     j=j*i;
```

Listing 2.5: Type IV - Original Code

This code snippet computes the factorial value of 'f', where 'j' is the result. Listing 2.6 realizes the same computation with a recursive function.

```
1 int factorial(int f) {
2     if (f == 0)
3         return 1;
4     else
5         return f * factorial(f-1) ;
6 }
```

Listing 2.6: Type IV - Code Clone

We can see that both fragments have the same behavior with equal input. For example: With $f = 6$ as input, the result of both computations will be 720. With no structural nor textual similarities, the code fragment has to be a *Type IV* clone.

2.1.1.5 Code Clone Terms

In addition to the four code clone types, there are other terms frequently used by clone detection software. The following section introduces important terms for further understanding.

Exact Clones: Exact clones are code fragments that are identical to each other, with the exception of changes in whitespace, layout and/or comments. An exact clone is basically referring to a *Type I* clone.

Near-Miss Clones: Near-miss clones are clones with changes in whitespace, layout, comments, and identifiers. That includes that all near-miss clones are *Type II* clones.

False-Positive Clones: False-positive clones are code fragments that are identified as clones. However, they are no code clones at all. Simple or repetitive structures often lead to these matches.

Semantic Clones: A semantic clone is another term for a *Type IV* clone. They are clones that are functional similar but are not necessarily syntactically similar.

Clone Pair: Two code fragments build a clone pair, if they are similar to each other.

2.1.2 Clone Detection Process

The aim of clone detection is to find fragments of high similarity. Due to the fact that it is not known beforehand which code fragments are used multiple times, the clone detector has to compare every possible fragment with each other. Even in smaller software systems this process consumes a high value of resources. To counter this problem, it is necessary to reduce the amount of possible fragments before performing the comparison. This section provides a brief description of every clone detection process step [RC07] [RCK09]. Figure 2.1 gives a graphical representation of the process.

1. **Preprocessing:**
   At first, the source code is partitioned and the source and comparison units are determined. This phase involves three steps.

   The first step is to *remove uninteresting parts* from the source code. This includes generated code(e.g., IDE generated UI parts), which may produce false-positives and embedded code from different languages(e.g, SQL embedded code).

   The next step is to *partition the remaining code into source units*. These source units are the largest fragments involved in the clone detection. Source units can be at any level of granularity, for example, files, classes, functions/methods, begin-end blocks, statements, or sequences of source lines.

   Finally, the *comparison unit* is determined. Depending on the actual clone detection technique (Section 2.1.3), the source unit is further partitioned into smaller parts(e.g., lines, tokens,..).

2. **Transformation:**
   If the clone detection technique is other than textual, the source code has to be transformed into the corresponding representation. Depending on the technique, the transformation is performed as :

   Tokenization: Each line of the source code is divided into small tokens depending on the rules of the programming language.

   Parsing: The source code is parsed into a parse tree or an abstract syntax tree.

   Generating PDG: Semantics-aware approaches generate program dependency graphs (PDGs) from the source code, where the source and comparison units are represented as sub-graphs.

   Some tools include a normalization phase during the transformation. The normalization includes: Removal of comments and whitespace, normalizing of identifiers or removing differences between layout and spacing - called pretty-printing.

3. **Match Detection:**
   The transformed code is analyzed during this phase. Every transformed comparison unit is compared to each other with the help of a comparison algorithm. Adjacent similar units are aggregated to form larger units. The result of this phase is a clone pair list.

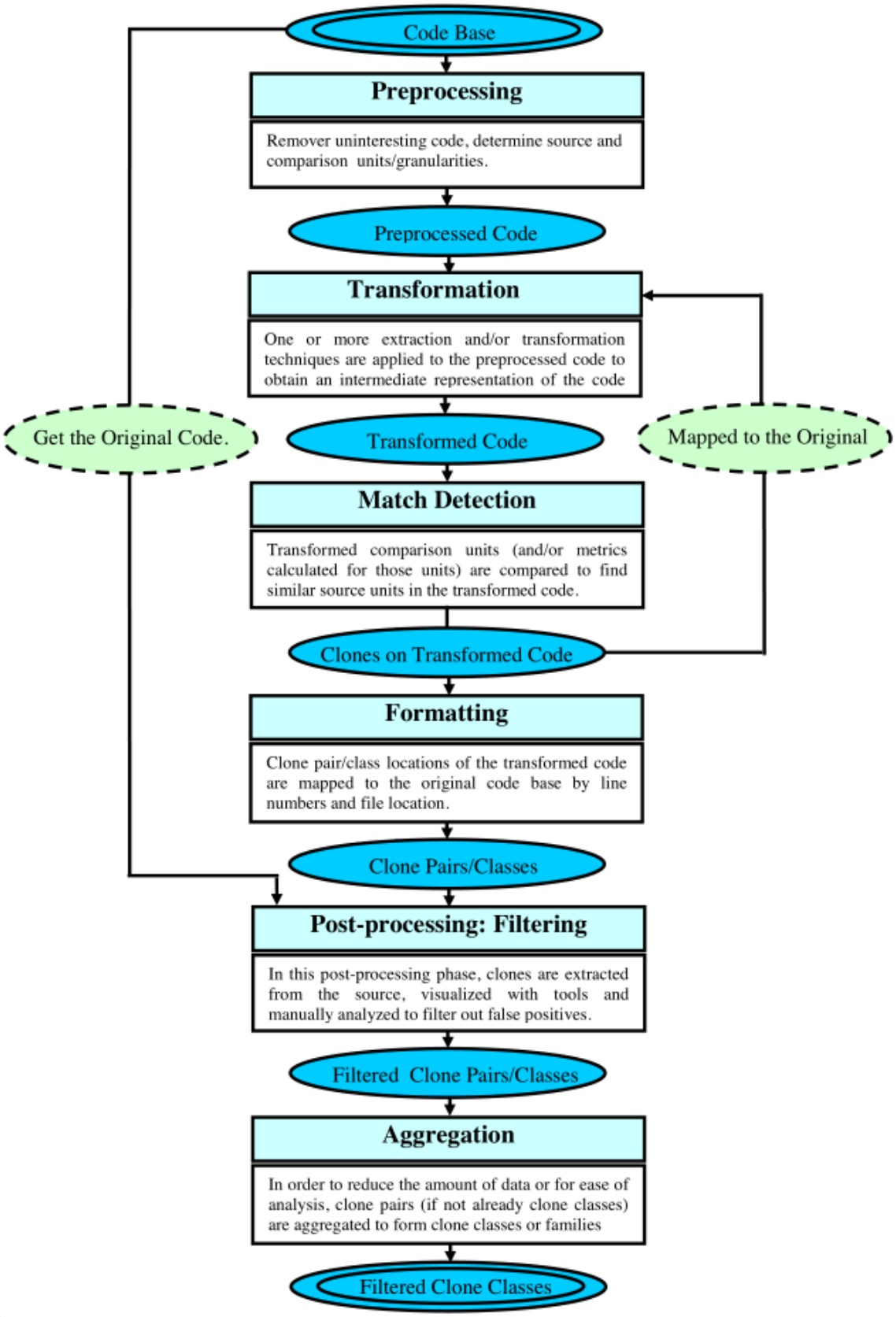

Figure 2.1: A Generic Clone Detection Process [RCK09]

4. **Formatting:**
   Because of the transformation, the clone pair list obtained by the comparison algorithm does not correspond to the actual source code. Therefore, the clone pair list is converted into a new clone pair list corresponding to the original code.

5. **Post-Processing:**
   False-positive clones are filtered out with the help of manual analysis or automated heuristics during this phase.

6. **Aggregation:**
   Finally, some tools reduce the amount of data by aggregation clone pairs into cluster, classes, groups etc..

2.1.3 Clone Detection Techniques

The following section introduces the basics of different clone detection processes. Furthermore, we give a short example of an actual implementation for each technique. Based on the information that is used during the process and the analysis technique, the techniques can be classified as one of the following main categories: *textual, token-based, tree-based or PDG-based.*

2.1.3.1 Textual

Textual approaches are based on the comparison of line or string sequences. While they mostly work with raw source code, some transformations or normalization can be applied such as comment and whitespace removal. Text-based techniques are not very robust against changes (e.g., structural changes such as the beginning/ending of a block), but on the positive side they are language independent. When two or more sequences are compared and identified as similar, the detector returns clone pairs with their maximum possible length. An example for a textual detector is the common `diff` file comparison utility, which tries to find differences between two files by finding the longest common subsequences.

2.1.3.2 Token-based

In a token-based approach, the source code is parsed into a sequences of tokens. Figure 2.2 is an example for transforming actual java source code into a token representation.The transformed sequences are then compared to each other to find duplicates. Once the clone pairs are found, the original code fragments are returned.

The token-based clone detector that is used for the experiment in this thesis is *JPLAG* by Pretchelt et al. [LP]. It is a java based software plagiarism detector, which is basically a cross-project code clone detector. The detection is split in two phases. In the first phase, the source code is parsed into tokens. Comments and whitespace do not produce a token, since they are easily changed. Every relevant code fragment is parsed into a program specific token (see Figure 2.2).

	Java source code	Generated tokens
1	`public class Count {`	BEGINCLASS
2	`  public static void main(String[] args)`	VARDEF,BEGINMETHOD
3	`            throws java.io.IOException {`	
4	`    int count = 0;`	VARDEF,ASSIGN
5		
6	`    while (System.in.read() != -1)`	APPLY,BEGINWHILE
7	`      count++;`	ASSIGN,ENDWHILE
8	`    System.out.println(count+" chars.");`	APPLY
9	`  }`	ENDMETHOD
10	`}`	ENDCLASS

Figure 2.2: Java Code to Token Example

The second phase is the detection phase. At first two token strings are compared to each other. The aim is to find substrings which meet the following requirements: token A can only be matched with exactly one token B. Substrings are to be found independent of their position in the string. Additionally, long substring matches are preferred over short ones.

The biggest contiguous matches are then marked, to exclude them from the comparison. This process is repeated until no more matches can be found.

2.1.3.3 Tree-based

A tree-based detector parses the source code into a parse tree or an abstract syntax tree (AST). The AST represents the syntactic structure of the parsed source code with abstracted representations for every element. Then, a tree matching algorithm is used to search for similar subtrees to detect the code clones. Once the clones are found, the corresponding source code of the subtrees is returned as clone pairs.

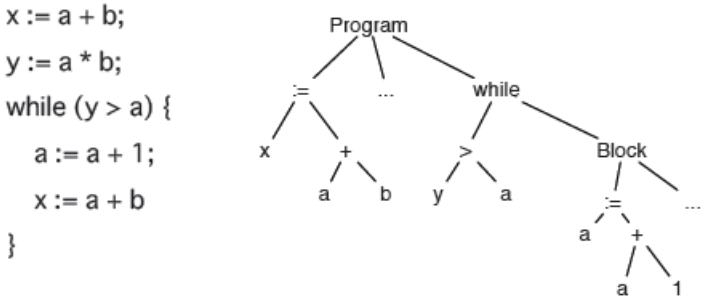

Figure 2.3: Java Code to AST Example

An example for a tree-based clone detector is *CloneDigger* by Bulychev et al. [BM09]. At first, the source code is parsed to an abstract syntax tree where each statement is

considered a root of its own subtree. Then, the detector identifies similar statements by using anti-unification (basically a generalization of statements [Rey70]). Similar statements are partitioned in clusters, where every cluster is marked by an ID. When the clustering is completed, duplicate clone fragments are found by finding identical sequences of cluster IDs. At last, identified code sequences are examined for overall similarity. In this phase, every pair of candidate sequences is checked for overall similarity at the statement level, again using anti-unification. When the statements surpass several thresholds, they are reported as a code clone pair.

2.1.3.4 PDG-based / Semantics-Aware

Semantic-Aware approaches aim at analyzing the behavior of the source code, rather than their syntactical similarities. A highly abstracted source code representation - called program dependency graph (PDG) is obtained that carries the semantic information of the source. The PDG contains the data and control flow, thus ignoring the syntactic structure. After obtaining the PDG, a subgraph matching algorithm is applied to discover similar subgraphs. These graphs are then returned as clones.

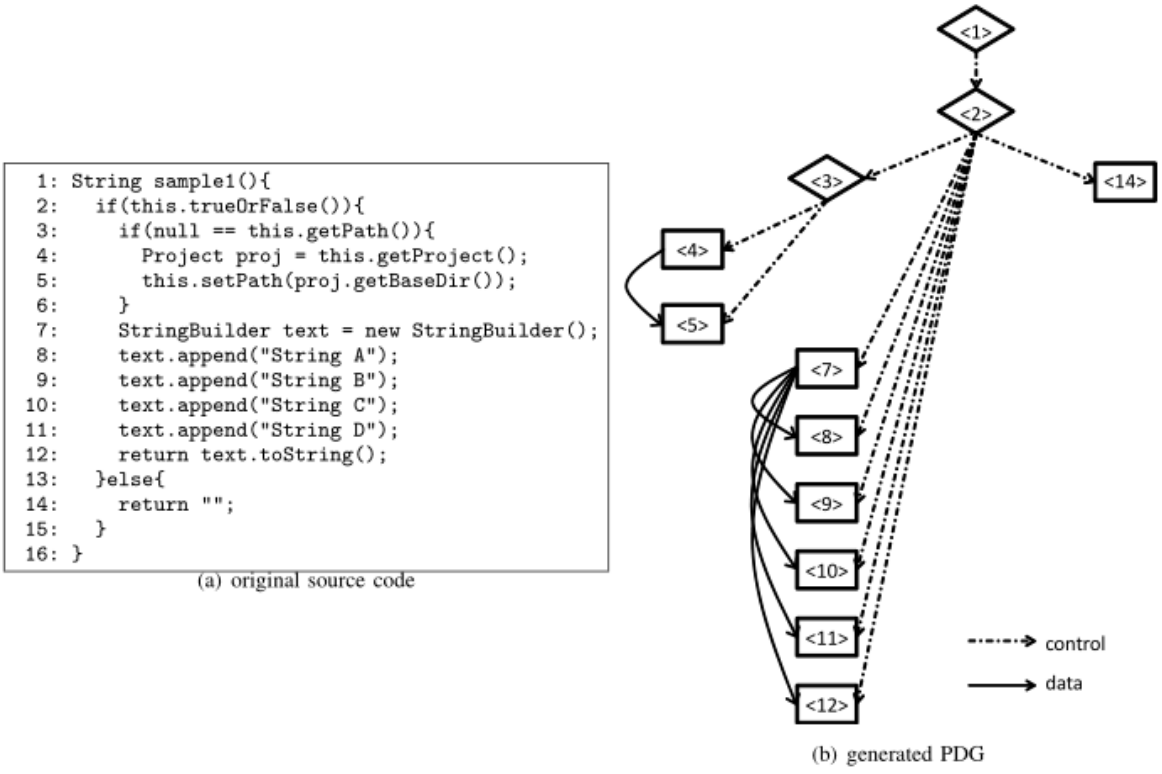

Figure 2.4: Java Code to PDG Example

A PDG-based clone detector is *Scorpio* by Higo et. al [HK11]. Its basic comparison algorithm consists of 4 steps after parsing the source code to a dependency graph. The first step is to hash all PDG nodes based on their contents. Nodes with similar hash values are classified as an equivalence class.

The next step is to find similar subgraphs. Therefore, a pair of nodes from one equivalence class is selected, which build the starting point for the comparison. If the predecessors (or successors) have the same hash value, then they are considered part of a similar subgraph. This is repeated, until the hash value of the predecessors(successors) is different. The pairs of identified similar subgraphs are clone pairs. In step 3, clone pairs that are subsumed by another clone pair are removed from the set, since they are part of a bigger clone pair. Finally, clone pairs sharing the same subgraph (e.g, $(s1, s2)$ and $(s2, 3)$ build the clonesets $(s1, s2, s3)$).

2.2 Code Obfuscation

During the development of a software system, a lot of know how of the developers is incorporated in the system. A software system is therefore a knowledge base for algorithmical problem-solving strategies. The compiled source code contains this knowledge. Unfortunally, there are certain (tool-assisted) possibilities to decompile every source code despite several protection mechanisms, thus enabling other developers to gain the encoded knowledge (illegally, if the source code is under copyright).

A creator of an application has different possibilities to protect his intellectual property. This includes legal security means such as copyrights or technical means such as server-side execution(application is running on a server, and the user never receives a physical copy) or encoding the program. Another possibility is to obfuscate(e.g., an extreme change of the layout) the source code, thus making the analysis more complex.

However, code obfuscation can be used the other way around as well. A competitor of a software developer can acquire the source code of an application by reverse engineering. He then develops a similar application using the acquired source code. Code obfuscation is then applied, to make it harder to find the similarities regarding the original program.

This section gives a short definition of code obfuscation and its terminology and shows a basic algorithm of code obfuscation.

2.2.1 Definition

Code Obfuscation is the process of changing the source code of an application while keeping its functionality. This may be done through means such as the changing of layout, renaming of identifiers, reordering, cloning of methods and much more.
Collberg et al. [CC97] defined an obfuscating transformation as follows:

DEFINITION (OBFUSCATING TRANSFORMATION)

Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{\tau} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*. More precisely, in order $P \xrightarrow{\tau} P'$ to be a legal obfuscating transformation the following conditions must be hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

Observable behavior in this context means the behavior the user experiences, not equal behavior. The result of P' has always to be the same as the result of P. However, P' does not have to be equally efficient and can be different during the execution for instance, regarding performance or memory consumption.

There is a variety of obfuscating transformations that can be applied to source code. Four groups have been proposed by Collberg to classify several obfuscating transformations. [CC97] Figure 2.5 shows some examples for the categories.

Layout Obfuscation: Layout obfuscations aim at making the source code unreadable. Operations of these categories do not require a lot of computational work but can easily get de-obfuscated. Obfuscating transformations that belong to this category are removing of comments, scrambling of identifiers and formatting changes.

Data Obfuscation: Data obfuscations affect the data and data structures. This may be done by splitting variables, promoting variables (i.e, int to integer object) or restructuring arrays.

Control Obfuscation: Control Obfuscations obscure the control flow of the execution. These transformations affect the aggregation (break up or merge statements), ordering (randomizing the order of statements) or computations (insertion of new dead-/redundant code, algorithmic changes) of a program. Because these transformations alter the overall control flow of the underlying program, a high computational overhead is necessary, but yields a high degree obfuscation.

Preventive Obfuscation: All aforementioned obfuscations can get detected and reverted by applications known as deobfuscators. Preventive transformations do not aim at obscuring the program to a human reader. Rather, they try to make known automatic deobfuscations techniques more difficult.

The quality of each transformation differs from each other depending on how good they obscure the program or how computationally intensive they are. An obfuscation can be evaluated by the following four criteria:

- **Potency:** Amount of obscurity added to the program by the obfuscation
- **Resilience:** Difficulty of how to break the obfuscation by an automated deobfuscator
- **Stealth:** Quality of how good the obfuscation blends in with the rest of the program
- **Cost:** Amount of computational resources consumed by the execution of the obfuscation

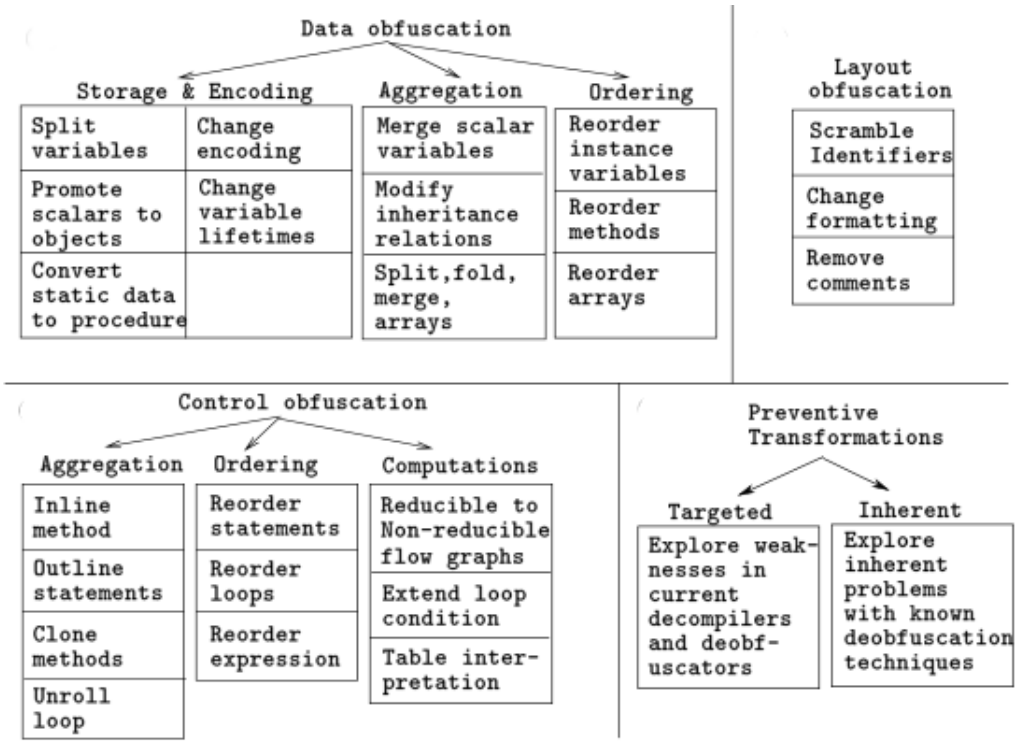

Figure 2.5: Classifications of Obfuscations [CC97]

2.2.2 Algorithms of Obfuscation

A general obfuscation algorithm was proposed in [CC97]. The top-level loop and general structure of an obfuscation tool is:

Algorithm 1 Basic Obfuscation Algorithm

while $!(Done(A))$ **do**
  $S := SelectCode(A)$;
  $T := SelectTransform(S)$;
  $A := Apply(T, S)$;
end while

SelectCode determines the next source code object(S) (e.g., a `for` statement) that is to be obfuscated. This object is then passed to the *SelectTransformation* function, that analyzes the given source code object and returns the transformation (T), which will be applied to S. At last, *Apply* performs the transformation T on the source code object S and updates the application. The algorithm terminates, when a specific event occurs - for example: no more transformations can be applied on the source code A, A exceeds a specific size or a certain degree of obfuscation is applied.

2.2.3 Applications of Code Obfuscation

Code obfuscation can be used for a variety of purposes by either legitimate software developers or malware authors. The main purpose of obfuscation is to protect software. As mentioned earlier, an obfuscated code will hinder the program to get reverse-engineered, thus saving its intellectual property. Moreover, code obfuscation can be used for watermarking by introducing special code - named 'markers'. As a result, granting the ability to clearly identify the version or the author of a program. Furthermore, code obfuscation can serve as a brain teaser - here known as recreational obfuscation. The participants aim is to either analyze an obfuscated program's function or obfuscate a given program as best as possible. [IOC]

On the other side, code obfuscation can be used for malicious intents, too. Virus creators can obfuscate their virus code, to create a new generation of their virus, thus hindering virus scanners to identify them as such. Moreover, Obfuscation may be used for concealing plagiarized software against plagiarism detection.

3. Framework

For the execution of the experiment different obfuscated versions of the the original source code of an application are required. They build the comparison set for the code clone detectors. Depending on the size of the source code, it would be very time-consuming and error-prone to obfuscate a whole Java project, if made manually. Therefore, we developed a simple code obfuscator as an Eclipse plugin, that applies different source code transformations that are intended to obfuscate the source code.

This section introduces the code obfuscator we developed for the experiment. The first part specifies some requirements the obfuscator has to meet. The second part describes the implemented transformations and classifies them. Finally, we present the overall program flow of the obfuscator.

3.1 Requirements

The aim of the code obfuscator is to provide a fast way to obfuscate any given source code automatically. To achieve this, several requirements must be met:

- **Eclipse Plugin:**
  The Eclipse IDE offers a variety of functions and frameworks which support the development of source-code based operations - most importantly the *Java Development Tools (JDT)* and the *Language Tool Kit (LTK)*. The JDT Core package provides the developer with APIs for navigating through the Java element tree and manipulating structured Java documents, thus supporting the developer with algorithms for the selection of code to be obfuscated. With the addition of the LTK package, which provides an API for refactoring, Eclipse offers the perfect environment for Java source code obfuscations.

- **Implementation of Obfuscations:**
  The code obfuscation can be split in 2 phases. The first phase is to find and select

potential source code fragments for obfuscation. The second phase applies the code obfuscation to the source code through code refactoring. These phases have to be implemented for every chosen obfuscation to work properly.

- **Selectable Obfuscations:**
  For the experiment, we need to be able to analyze the effect of every single obfuscation. If the source code is obfuscated with all obfuscations at once, we are not able to identify the effect of them separately. Therefore, the obfuscator must have the ability to let the user select, which obfuscations he wants to apply.

- **Syntactically-correct Transformations:**
  The definition of obfuscating transformations demands, that the observable behavior of the obfuscated program in comparison to the original program needs to be the same. Furthermore, the obfuscated program must terminate and produce the same output as the original. Therefore we need the implemented transformations to be syntactically correct regarding the Java language and the outcome of the original program.

- **Logging:**
  For the evaluation of each clone detection result, it is necessary to know where code was obfuscated and the amount of obfuscations performed on the original source code. Additionally, the logging allows to build a correlation between the code clone detection results and the performed obfuscation. Therefore, we need a logger for creating a log file which contains all necessary data of the obfuscation process.

- **Simple Interface:**
  The obfuscations do not need extensive user-input except for selecting active obfuscation. Therefore, a simple-user interface (status overview, checkboxes, ...) is sufficient for our purposes.

3.2 Transformations

For the purpose of the experiment, we need viable transformations to test the robustness of the clone detectors. Therefore, we decided to use simple transformations that can be done by anyone, for example a student who is trying to obscure a program for an assignment which is not his own work. The used transformations are presented in this section. A summarized view of the transformations is presented in the appendix (Chapter A).

3.2.1 Renaming

The renaming transformation changes user-defined identifiers (fields, variables and methods). Without any user input, the identifiers are named sequentially ($m1, m2, ...$ for methods, $v1, v2, ...$ for fields/variables). However, the user can define new names by

himself. Following the definition of code clone types, a *Type II* clone is created by the transformation, while the transformation itself is a layout obfuscation. We present an example for such listing in Listing 2.1:

```
1  class foo{                                  class foo{
2
3  public int a = 0;                           public int v1 = 0;
4  public int testMethod(int b) {              public int m1(int v2) {
5      return int c = a + b;                       return int v3 = v1 + v2;
6  }                                           }
```

Listing 3.1: Layout Obfuscation using Renaming

3.2.2 Expansion

Basically, the assignment operators(+=, -=, *=, /=) are eliminated but their logic and semantics will be reconstructed. These new statements contain changes compared to the original or new statements are introduced. Consequently, the original and the transformed code fragment form a *Type III* clone. Moreover, this transformation is a control obfuscation. The basic replacements are the following:

```
1      ...
2      i++;          i = i + 1;
3
4      a += i;       a = a + i;
5
6      a = ++i;      i = i + 1;
7                    a = i;
8
9      a = i++;      a = i;
10                   i = i + 1;
11     ...
```

Listing 3.2: Control Obfuscation by Expansion

3.2.3 Contraction

Contraction is the opposite to the expansion obfuscations. Instead of eliminating assignment or increment/decrement operators, this operation converts possible statements to the shortened statement. Similar to the expansion obfuscation, it creates a *Type III* clone and is a control obfuscation. Basic replacements are presented here:

```
1  ...
2  i = i + 1;        i++;
3
4  i = i + x;        i += 0 + (x);
5  ...
```

Listing 3.3: Control Obfuscation by Contraction

3.2.4 Loop Transformation

The loop transformations aim at swapping `for` and `while` loops. It creates a *Type III* clone, but since for and while loops are treated differently, it can be regarded as a *Type IV* clone. The obfuscation is a control obfuscation. The following is an example of a conversion from `for` to `while`:

```
1 for(int i = 0; i < 20; i++} {          int i = 0;
2     \\Execution block                  while(i < 20) {
3 }                                          //Execution block
4                                            i++;
5                                        }
```

Listing 3.4: Control Obfuscation by Loop Transfomation I

The backwards transformation is presented in listing 3.5.

```
1 int i = 0;                             int i = 0;
2 while(i < 20) {                        for(;i < 20;)
3     //Execution block                        //Execution Block
4     i++;                                   i++;
5 }                                      }
```

Listing 3.5: Control Obfuscation by Loop Transfomation II

3.2.5 Conditional Transformation

In Java, an if-else condition statement can be converted to a shortened expression statement, if both blocks contain an assignment with the same variable on the left. The conditional transformation performs this conversion. Furthermore, it converts the shortened form to the standard if-else statement. Hence, this transformation creates a *Type III* clone and is an control obfuscation.

```
1 if (a == b)      a = (a == b) ? b : 0;
2     a = b;
3 else
4     a = 0;
```

Listing 3.6: Control Obfuscation by Conditional Transformation

3.3 Obfuscator Implementation

In the last section we presented the obfuscations the obfuscator is capable of. In order to apply them practically, we developed an Eclipse plug-in called *ARTIFICE*.

The *ARTIFICE* obfuscater plug-in consists of four main parts - user-interface, logging unit, data-converter and refactoring unit. The interface handles all user-input (activating/deactivating obfuscations), while the logging unit logs all obfuscations made and gives a summary of the whole process. The data-converter creates renaming jobs (e.g.,

a bundle of method and new method name) prior to the obfuscation process for more customization possibilities. Finally, the refactoring unit performs two tasks: finding possible code fragments for all obfuscations (except renamings) and performing every obfuscation through refactoring.

In Figure 2.1, we show an overview of the general process of our tool. In the following, we present detailed internals for the different phases of this process.

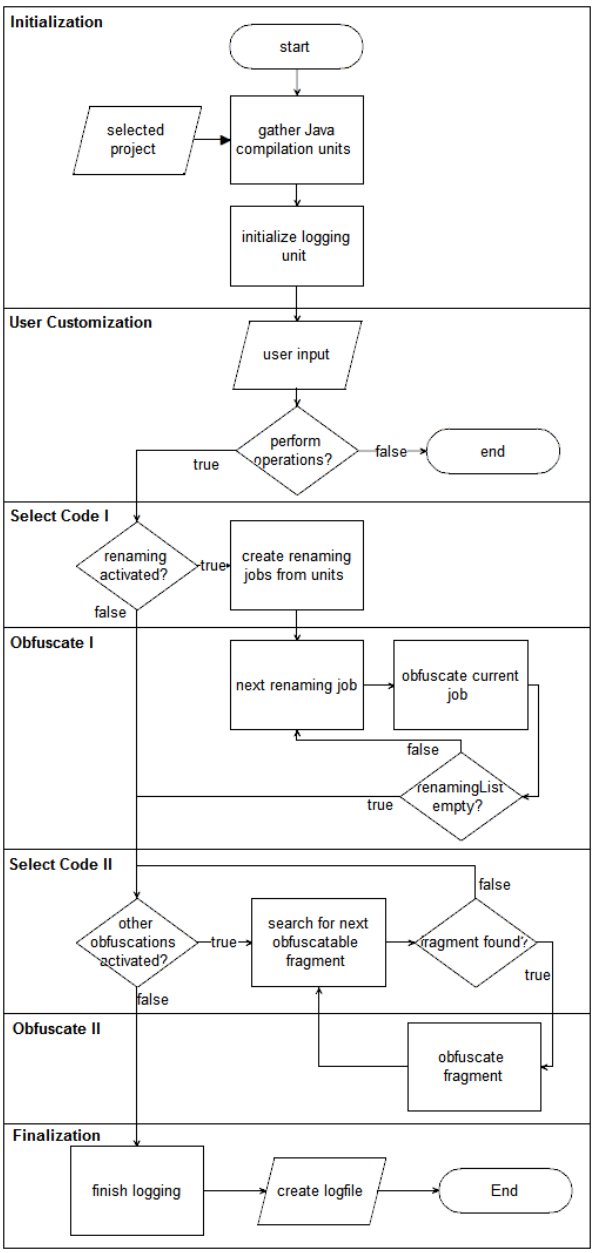

Figure 3.1: ARTIFICE - Program Flow Chart

1. **Initialization:** At the beginning, the user has to select a Java project in the project explorer of Eclipse. This project contains all Java files that get obfuscated during the process.Then, these files are saved in map containing the compilation unit (represents the *.java* source file) of the file as key ,and a *renamingList* as value. The *renamingList* will contain every method, field and variable with their respective new name. This data is created in *Select Code I*. Finally, the logging unit is initialized for further use.

2. **User Customization:** After the initialization of necessary parts, the program opens a dialog where the user can customize specific points. Among others, these include the activation/deactivation of obfuscations and customizable names for every renaming obfuscation. After finishing the customization process, the actual obfuscation starts.

3. **Select Code I (Renaming):** This step handles the first part of code selection. Here, the data converter searches for specific elements offered by the JDT Core package [JDT]. The package defines the classes that compose a Java program. It uses a hierarchical model that represents the structure of a Java program (Java Model).

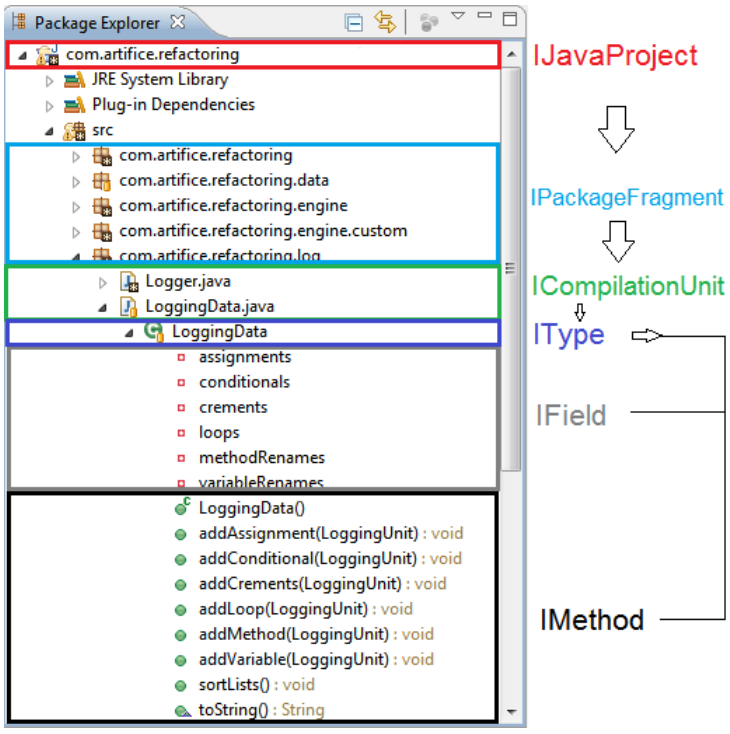

Figure 3.2: Java Model Example

Starting from the root of the model ($IJavaModel$), the program walks through the hierarchy to find every $IField$ for fields and $IMethod$ for methods. However, $ILocalVariable$ is no part of the Java model, therefore the program analyzes

the program's AST to find variables. This information is then used to create a *renaming job*. The job is build from the corresponding *IJavaElement* and a corresponding new name. When every compilation unit has been searched, the program proceeds to the next part.

4. **Obfuscate I (Renaming):** With the information saved in a *renaming job*, we can use the renaming function of Eclipse to perform the obfuscation. Every necessary condition check is covered by Eclipse, as well as a project-wide transformation. Furthermore, every obfuscation is logged by the logging unit.

5. **Select Code II:** The next step is to apply the other obfuscations, if at least one of them has been activated by the user. For selecting the code, the program searches for AST elements corresponding to the selected transformation (e.g., WhileStatement or ForStatement for the loop transformation). Then, the surrounding elements are examined. If they allow the specific obfuscation, the AST element can be obfuscated.

   When the obfuscation of this fragment is complete, the next AST element is examined. It is important to mention, that the refactoring is *not* performed on the currently loaded compilation unit. In fact, a single refactoring is performed on an image of the original source code. If no errors occurred during the process, the original compilation unit is overwritten by the image. Since an element of the Java AST is dependent on its offset and length, we have to consider that a single change will influence every following element. Consequently, we have to reload the compilation unit after every successful refactoring. Only then the next code fragment can be selected.

   In case that no more corresponding code fragments for the current obfuscation can be found, either the next activated obfuscation is handled with or the process is finished.

6. **Obfuscate II:** Here, the previously selected code is committed to a customized refactoring process not provided as a standard eclipse refactoring function. First, to allow the obfuscation, some conditions have to be checked such as if the compilation unit is error free. Then, the AST of a parent node of the edited AST node forms the starting point. Then, an image of this AST is created. The editing is applied to this image first. When the transformation of the image is completed, the change is applied on the actual AST. Finally, the file is rewritten based on the new AST. The obfuscation is now applied. During this, the code is changed based on the obfuscation (Section 3.2) and the change is logged by the logging unit.

7. **Finalization:** After every obfuscation process is completed, the logging unit stores the information gathered throughout the process into a log file. The obfuscation is now done.

4. Evaluation

The Oxford Dictionary defines *robust* as the ability *"to withstand or overcome adverse conditions"*. *Robustness* is the quality of being robust. In computer science, robustness is the ability of a computer system to cope with errors during execution or the ability of an algorithm to continue to operate despite abnormalities in input, calculations, etc.

The input for a code clone detector is the source code of a system. With the general purpose of code clone detectors in mind, we can assume that the detector will find a code clone even if a specific fragment is changed (the abnormality). Therefore, the robustness of a code clone detector is the ability find code clones despite changes, that doesn't result in syntactical changes (obfuscations).

With the following case study, we want to evaluate the robustness of each code clone technique regarding our presented obfuscations. First, we describe the setup of our study and the methodology to assess the robustness of detection techniques against code obfuscations. Afterwards, we present our results and interpret them. Finally, we discuss threats to validity that may limit the results of our study.

4.1 Setup

At first, and maybe most important requirement, we need a comprehensible subject software system, suitable to demonstrate the functionality of our obfuscation. Therefore, we used a remake of the classic *PacMan* game as test subject. The game works as follows: The player controls the pacman with the aim to gather all white dots in a maze, while being followed by four ghosts. If a ghost touches the pacman the game ends. If the player gathered a bigger white dot, the ghost flee from pacman and he can destroy them. The ghosts respawn after a little while.

For the detection process we used the following detectors: *JPLAG* (v.2.2.1) is used for the text- and token-based detection. It is widely known and a reliable plagiarism

detector tool, that is able to detect clones in a variety of languages (Text, C, C++, C#, Java).
CloneDigger (Revision 211) is used for the AST-based clone detection, which uses anti-unifaction([Rey70]) to find clones among Python and Java source code files.
At last, we use *Scorpio*(v201103030634) for the PDG-based detection. *Scorpio* uses a modified program dependency graph with an additional execution edge between two nodes, to overcome some limitations of PDG-based clone detection.

4.2 Methodology

For the analysis of the robustness, we need a comprehensible, reproducible testing method. Therefore, we used the following way to conduct the experiment:

- Preparation: Before we can use a clone detector we need obfuscated copies of a original program. Therefore, we used the code obfuscator to create these copies, where every possible combination builds a new obfuscated copy. A copy with a single applied obfuscation will give a direct insight of the robustness regarding the specific obfuscation. Combined obfuscations will show if the detection rate decreases linear (results add up) or if the detector is more vulnerable regarding certain combinations of obfuscations at the same time.

1. At first, we provide a (theoretical) reasoning about the robustness of the specific technique. With the knowledge of the functionality of the technique, we can estimate how much impact the obfuscation may have on the detection rate. Later, we will compare our presumption with the actual results.

2. The first test is to use the clone detector with the original and a copy of the original. The detection rate between these two will likely result in a 100% percent match. If the detection tool does not give the option to disable clone detection inside a project itself, further analysis is necessary to find groups of code clones that can be ignored. We use the result of this initial clone detection as reference values for all other comparisons, because this allows us to evaluate the relative change impact, an obfuscation may have on the detection technique. Depending on the tool it may offer different values such as number of code clones or lines of cloned code. This step is necessary because we need a numerical value for the comparison to rate the impact of an obfuscation.

3. Each obfuscated copy is then checked by the clone detector with the original as reference.

4. Then, the results of each detection process will be presented in a summary table. The name of each obfuscation step corresponds to the obfuscations applied on the code (R - Renaming, E - Expansion, C - Contraction, L - Loop Transformation, I - Conditional Transformation).

5. The last step is to evaluate the results regarding its robustness and compare it to the initial reasoning.

The software systems consists of 21 source files with 2415 lines of the code. The original source code is copied sixteen times (one for each obfuscation combination), on which we used the described *ARTIFICE* obfuscator. The obfuscator was able to apply 616 renaming obfuscations (120 methods, 155 fields, 341 variables), 54 expansions, 8 contractions, 34 loop transformations and 0 conditional transformations. Every obfuscated copy was tested for full functionality, to ensure that the obfuscations didn't introduce errors and didn't alter its behavior.

4.3 Results

In the following, we present the results for every clone detector. Based on these results, we discuss and assess the robustness of each detection tool. Furthermore, we put the result in the context of the actual, underlying detection technique and discuss its influence on the robustness we measured.

4.3.1 Robustness of Text-Based Clone Detection

Text-based code clone detectors rely solely on the textual representation of the source code. Only minor transformations are performed, such as whitespace, comments and layout is normalized. This makes is difficult for the clone detector to detect *Type II, III and IV* clones. Therefore, we assume that every obfuscation has an impact on the detection rate of the detector. For instance, renaming will affect the clone detector the most since it affects nearly every line of code. We assume that the robustness of a textual clone detector is generally low.

For our analysis, we used *JPLAG* as our textual clone detector. Although, it is used as a Java token-based clone detector, it has an integrated text clone detector as well. *JPLAG* presents its result in different ways: A side-by-side file comparison with highlighted code clones and a similarity value in percent between two files. Additionally, the detector counts the string tokens of each code clone, which can be used to describe the size of a code clone. Adding each value of each code clone up, will result in the size of cloned code. Therefore it is possible to calculate the number of code clones and a value similar to cloned lines. These values strongly relate to each other. The more code clones exist, the less string tokens are contained in a code clone, since one code clone ends when a difference between two files is found. The next characters are not considered as a code clone until the minimal match length is reached. Only then a next code clone can be found. The smaller the size of code clones is, the smaller is the similarity. In Figure 4.1, we can see that a clone is finished when there is a difference between these two fragments and the next string tokens are not part of a code clone. A new code clone is found after the difference. Therefore, the code clone amount is increased by one, but the size of code clones is decreased.

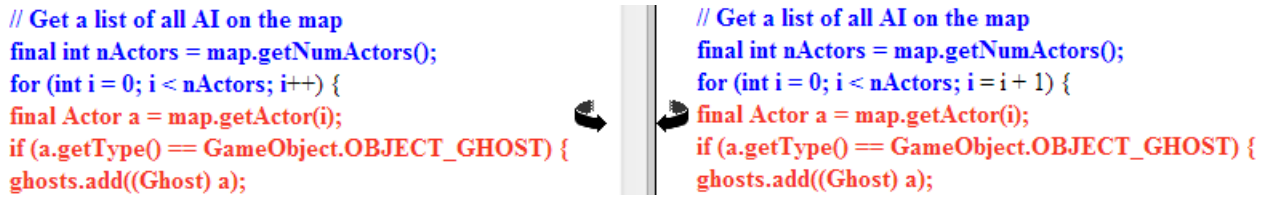

Figure 4.1: JPLAG (detected clones are highlighted)

We used the default settings for *JPLAG*. Hence, the minimal match length is set to 5 string tokens. Smaller similar strings are not considered a code clone. We present the result of the clone detection before and after obfuscation in Table 4.1.

Obfuscated Project	Code Clone Amount	Size of Code Clones	Similarity in %
Original	21	14513	99,8
C	29	14505	99,8
E	68	14497	99,3
EC	74	14481	99,2
L	96	13875	96,7
CL	104	13867	96,7
EL	113	13860	96,2
ECL	119	13844	96,1
R	608	8461	59
RE	608	8461	59
RC	608	8461	59
REC	608	8461	59
RL	576	8046	56,6
REL	576	8046	56,6
RCL	576	8046	56,6
RECL	576	8046	56,6

Table 4.1: Clone Detection Results - Textual (Match-length 5 (String Token))

As we can see, the comparison between the original and a non-obfuscated copy did not score 100%. The side-by-side view reveals, that the beginning of every java file (`package` declaration) and the last block ending brackets are not considered as part of a code clone. This may be an internal error that creates the deviation.

The contraction obfuscation has the same similarity as the original comparison. However, we can see an increase in code clone amount and accumulated size of the code clones. Since there are only eight contractions applied, it disturbed the detector pretty well. Eight new code clones were found and the size decreased by eight, which means that all obfuscations influenced the detector.

JPLAG behaves similar with the expansion obfuscations. A decrease of 0.5% in similarity and 47 more code clones were found. There were more expansions than contractions, and as a result they had more impact on the result.

The combination of contraction and expansion resulted in nearly combined values. The size decreased by sixteen, double of the amount of contractions. However, only six more code clones where found. Due to the minimal match length, some strings were not considered part of a code clone, because two obfuscations were surrounding the part.

The loop transformation resulted in a decrease of 3.1% and an increase of 75 more code clones. Additionally, the size decreased by 638. Considering the small amount of loop transformations, the code clone detector could not identify the similarity. The transformation from `for` to `while` changed the whole initialization of the loop, and thus has a considerable impact on the detection rate. The combination with expansion, contraction or both added little obfuscation, since a lot expansions were made to the updater of the `for` loops. These fragments were already affected by the loop transformation.

As expected, the renaming obfuscation had the highest impact. The similarity decreased by 40.8%, 508 more code clones were found and the size of the code clones decreased by 6052. A huge part of the code is affected by the renaming, since every statement contains variables or fields. The combination with expansion or contraction did not change the result, since these code lines are already affected by the renaming obfuscation. With the addition of the loop transformation, the code clone amount decreases but the size decreases as well. This is a result of the minimal match-length, since more code is not part of a code clone.

In summary, we can say that the results correspond with our assumption at the beginning. Every obfuscation decreased the ability of the code clone detector to find code clones, with the renaming obfuscation as the best obfuscation. However, renaming obfuscation can be detected, if the detector normalizes identifiers.

Overall, the robustness of a textual-based clone detector can be considered very low, since every obfuscation decreased the similarity between the original source code and die obfuscated code.

4.3.2 Robustness of Token-Based Clone Detection

The token-based detector parses the whole source code and works on a token sequence as representation of the code. During creation of this token sequence, identifiers, whitespace and layout are normalized. Therefore, the clone detector should be able to detect *Type I and II* clones. However, *Type III and IV* clones represent a difficulty for the clone detector, since they interrupt a token chain. We assume that renaming has little to no effect on the detection rate, because of the normalization of identifiers. The other obfuscations should have a detrimental effect on the clone detector, since they change statements to a different appearance, which will result in different tokens.

Again, we used *JPLAG* for the detection. This time however, the source code is analyzed with the token-based technique. The size of code clones is calculated by adding the amount of tokens in each code clones up. For this detection process, we used the default settings with the minimal match-length set to nine tokens. We present the results of the clone detection in relation to the obfuscations in Table 4.2.

Obfuscated Project	Code Clone Amount	Size of Code Clones	Similarity in %
Original	20	3073	99,8
C	20	3073	99,8
E	20	3073	99,8
EC	20	3073	99,8
R	38	2973	96,6
RC	38	2973	96,6
RE	38	2973	96,6
REC	38	2973	96,6
L	45	2821	91,6
EL	45	2821	91,6
CL	45	2821	91,6
RL	45	2821	91,6
RCL	45	2821	91,6
ECL	45	2821	91,6
REL	45	2821	91,6
RECL	45	2821	91,6

Table 4.2: Clone Detection Results - Token (Match-length 9 (Java Token))

The comparison between two original copies of the source code resulted in a similarity of 99.8%. One java source file contained only four tokens and thus, the detector was unable to detect the clone with the default settings, since the file was too small. Consequently, we use the values of this comparison as reference values.

The expansion and contraction obfuscations or the combination of both did not alter the result of the detection process. *JPLAG* parses each assignment into an `ASSIGN` token. Therefore, the detector was able to identify the similarity between the original and the obfuscated source code.

The renaming obfuscation resulted in an increase of 18 code clones, while the size of code clones decreased by 100 tokens and the similarity by 3.2%. As presented in the appendix (Chapter A), the renaming obfuscation always splits variable declaration and initialization. Normally, the detector is able to find the similarity, since both before and after the obfuscations the tokens remain the same (before: `VARDEF ASSIGN;`, after: `VARDEF; ASSIGN;`). However, the same has to be applied to the initializer of a `for` loop, thus changing the order of the tokens. `BEGINFOR(VARDEF ASSIGN;...)` is the standard order, but the renaming obfuscations changes the order to `VARDEF; BEGINFOR(ASSIGN;...)`. Consequently, a clone always ends with every `for` loop, since the

tokens doesn't match at that point and thus decreasing the similarity. A combination with expansion or contraction does not alter the result.

The results of the comparison between the original program and the program after loop transformation shows an increase of 25 code clones, with a decrease in size by 252 tokens and a decrease in similarity by 8.2%. Both, the transformation from `for` to `while` as well as the backwards transformation interrupted code clones.

```
1 for(int i = 0; i < 10; i++){            BEGINFOR, VARDEF, ASSIGN, ASSIGN
2     .. code clone ...
3 }                                       ENDFOR
4 ──────────────────────────────────────────────────────────────────────────
5 int i = 0;                              VARDEF, ASSIGN
6 while(i < 10) {                         BEGINWHILE
7     ... code clone ...
8     i++;                                ASSIGN
9 }                                       ENDWHILE
```

Listing 4.1: Loop Transformation with Tokenization

Listing 4.1 shows an example of such transformation with the respective tokenization. If the code fragment, enclosed by the loop, exceeds the threshold for minimum clone length, it is detect as a code clone. However, the beginning and the end of both loops are not considered part of the code clone. We can see that *JPLAG* uses different token for each loop (beginning and end), which results in a difference between the code files and thus ending a code clone. Furthermore, the reordering of the initialization and the updater result in bigger difference.
A combination with expansion, contraction and renaming does not result in a different outcome. The changes that altered the results in the renaming/original comparison, are applied in the loop transformation, too.

Overall, the token-based approach performs better than expected. The renaming obfuscation is detected (except in a `for` loop). Additionally, the expansion and contraction obfuscations had no effect on the clone detector. However, more complex obfuscations such as the loop transformation represent a problem for *JPLAG*, since it is very dependent on the specific tokens. Even if two tokens represent a similar procedure, it is not capable to identify the similarity.

In summary, we argue that token-based clone detection is very robust against simple obfuscations, that target one line. An obfuscation that affects a larger code fragment of multiple lines however is a problem hence, the detector is not robust against them.

4.3.3 Robustness of PDG-Based Clone Detection

Instead of relying directly on the source code, the pdg-based code clone technique analyzes the program and data flow. The obfuscations made by the obfuscator create *Type II and III* clones. The loop transformation can be seen as creating a *Type IV* clone.

Because PDG-based clone detection uses a program dependence graph as representation, it should have the same detection rate as if no obfuscations were applied on the code. However, the loop transformation may have an impact.

We used the PDG-based tool *Scorpio* for the detection [HK11]. The detector combines similar code fragments into a cloneset, which consists of the description of every found clone (number of PDG nodes, location,..). This information is presented in a side-by-side view to analyze each code clone. With the knowledge of the number of nodes and the amount of code clones (min. two) in a cloneset, we can calculate the size of code clones. The number of nodes is equal for all clones and thus the size is calculated by multiplying both values with each other.

Since *Scorpio* does not offer the possibility to detect clones between projects only, we need to analyze which clonesets consists of at least one clone in the original source code and one clone in the obfuscated source code. To this end, the cross-project clones are identified programmatically by analyzing the XML result file made by *Scorpio*.

Generally, applying an obfuscation may lead to one of the following two results. First, there is the possibility to eliminate a cloneset after obfuscation, that normally contains two clones of two systems. The obfuscation changes one clone in the obfuscated project and the detector is not able to detect this clone again. Therefore, the cloneset would only consist of one code fragment, thus making it no clone at all. Additionally, only cross-project clones are relevant in this case study. Let's consider a cross-project cloneset consisting of three code clones(e.g, two in the original code and one in the copy of the original). After the obfuscation the detector can not find the clone in the copy, thus reducing the number of clones in the cloneset to two. However, these two clones are only in the original code. Consequently, the cloneset is no longer a cross-project cloneset, thus making the cloneset irrelevant for the case study. Figure 4.2 is of visual representation of this principle.

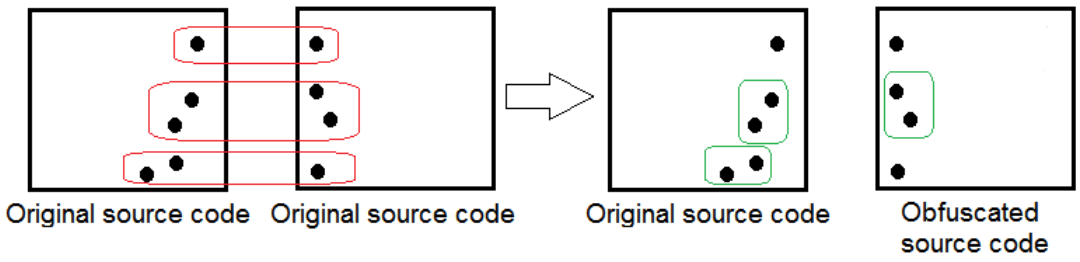

Figure 4.2: Cross-Project Clones (Red = Cross-Project, Green = Internal)

We used a minimal match-length of five PDG nodes (5 statements), because it created satisfactory results in comparison to other match-lengths. Furthermore, merging of statements is activated. This merges consecutive duplicate statements to one PDG node. Other settings are set to default, as seen in its description [SC]. We present the results of our clone detection in Table 4.3

Obfuscated Project	Total Clones		Cross-Project Clones		
	Clonesets	Size of Clones	Clonesets	Size of Clones	Similarity in %
Original	117	3333	79	2601	100
C	118	3290	71	2466	94.8
L	120	3232	76	2326	89.4
E	114	3142	71	2242	86.2
EL	118	3161	71	2232	85.8
CL	125	3215	68	2180	83.8
EC	117	3120	65	2134	82.0
ECL	117	3120	65	2134	82.0
R	143	3159	54	1383	53.2
RL	143	3128	53	1366	52.5
RC	137	3072	48	1316	50.6
REL	141	3118	49	1308	50.3
RCL	137	3080	45	1286	49.4
RECL	133	3020	45	1268	48.8
RE	133	2922	45	1164	44.8
REC	133	2910	41	1120	43.1

Table 4.3: Clone Detection Results - PDG (Match-length 5 (PDG Node))

The reference value for the following results is the size of clone value of the original detection process, which resulted in 2601 PDG nodes that are part of cross-project clonesets.

The contraction obfuscation shows a decrease of 135 nodes, despite the small amount of obfuscations applied. A deeper analysis, shows that the majority of contractions affected clonesets with only 5 PDG nodes, which is the minimal match-length. With the applied obfuscation, the clone only consists of four nodes that are part of a code clone and thus it is not identified as a clone. However, even if the match-length is set to four, the obfuscation still has an effect on the size, since at least one PDG node is missing. Consequently, the contraction creates a different data flow. The combination with other obfuscations shows a similar effect on the size of clones.

The loop obfuscation decreases the size of code clones by 275 PDG nodes. Considering the complexity of the transformation and the number of applied obfuscations, it is a rather small effect on the result. Generally, most of the transformations are detected as part of a code clone. However, the transformation from `for` to `while` is partially detected only. The updater of a `for` loop is the last statement of the `while` loop block after the obfuscation. The updater is part of a code clone before obfuscation. However, after their obfuscation it is not. This is shown in Figure 4.3. Consequently, it results in at least one less PDG node. If the code clone is smaller than the match-length after obfuscation, it is not a clone at all.

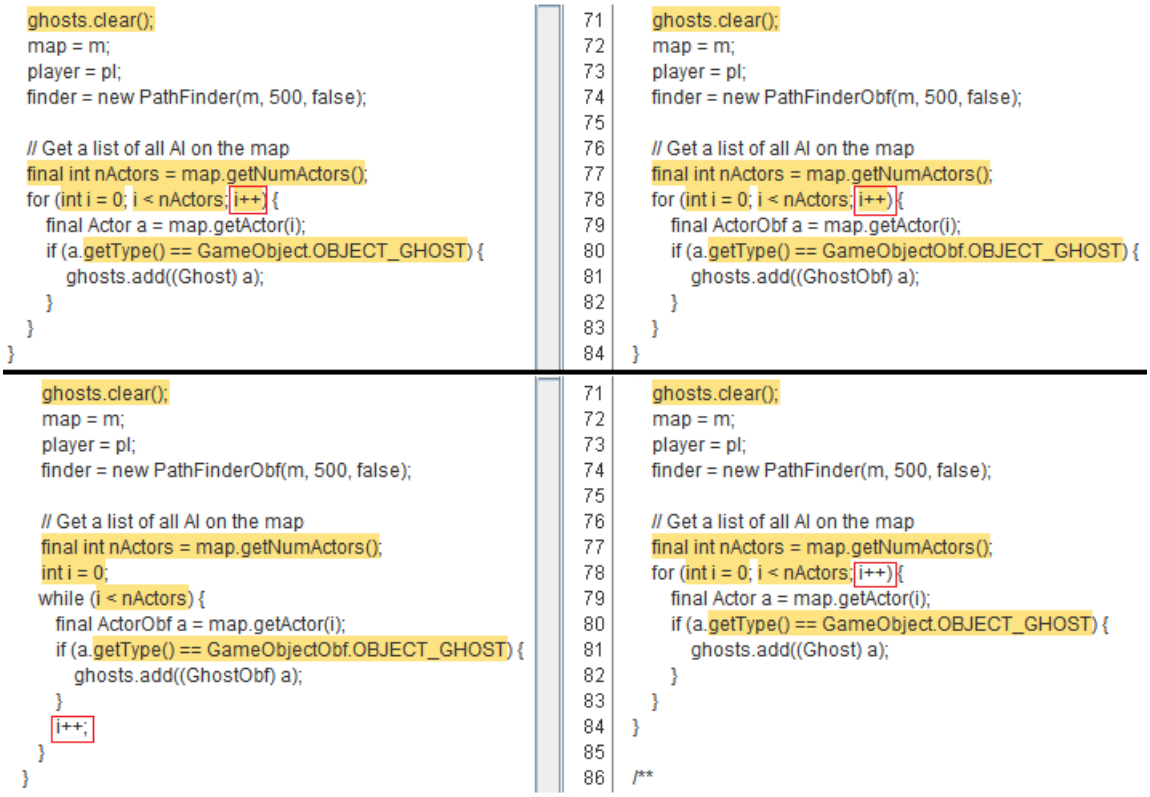

Figure 4.3: Scorpio - Loop Obfuscation Results (Top: Original/Original, Bottom: Obfuscated/Original) - Cloned PDG nodes are highlighted

A combination with another obfuscation shows little to no effect. However, sometimes the result is better for the detector with the loop transformation applied (RE vs. REL). This is a result of the `while` to `for` transformation. The boolean expression of a loop is part of a code clone before and after obfuscation. The initialization of a loop counter is not detected as part of a clone in `for` loop. However, it is part of a code clone with the transformed `while` loop, resulting in an increased size of code clones.

The expansion shows a considerable effect on the detection rate of *Scorpio*, by decreasing the detected size of clones by 359. The obfuscation transforms increment/decrement statements to assignment statements. These assignment statements are not part of a code clone after obfuscation, which decreases the size by either one or the code clone does not match the minimal match length, thus making it no code clone at all.

Renaming should have no effect on the detection rate. But as a matter of fact, the results show that this obfuscation has even a more observable influence on the detection result than each single obfuscation. In particular, this transformation decreases the size of clones by 1218 PDG nodes. The analysis of the results shows, that the detector has no problem with single renamings on one PDG node and such nodes are still part of code clone. However, when a node is increasingly affected by the renaming obfuscation, it loses its status as part of a code clone. The result is, that a huge amount of clones are

either reduced or not a code clone at all, thus declining the size of code clones. The combination with expansion, further decreases the size of clones.

Surprisingly, every obfuscation affected the size of clones, with renaming as the most effective one. The only obfuscation with little to no effect is the loop obfuscation.

The result does not match the with our assumption at the beginning. In the paper which presents the algorithm of *Scorpio* [HK11], it is mentioned, that PDG based clone detection is not very suitable for detection contiguous clone detection, whereas other technique perform better in such environment. This can be an explanation of the results. Primarily, PDG based technique aim at detecting special clones, for example the loop obfuscation was detected fairly well.

4.3.4 Robustness of Tree-Based Clone Detection

The tree-based detector uses an abstract syntax tree as work object. Similar to the token-based approach whitespace, layout and identifiers are normalized. *Type I and II* clones should be detectable by this clone detector. Also, the loop transformation could be easily detected, since the AST representation of a for and while loop are very similar.

For this analysis, we used AST-based clone detector *CloneDigger*. It creates a HTML file for the results, where each found clonepair is presented. The difference between each clone in a clone pair is highlighted by the use of the `diff` tool. Furthermore, the size of each clone and the number of clones are calculated.

Since *CloneDigger* analyzes source code files for internal clones, we need to consider the crossproject clones (discussed in Section 4.3.3). To find the influence of the obfuscations, we can compare the code of clonepairs between the original result with the obfuscated.

The minimal match length is set to 10 AST-nodes with a maximal distance of 200. *CloneDigger* ends a clone, if the differences between each clone of a clone pair exceeds the distance. Considering the amount of obfuscations done by the renaming obfuscation, a clone is ended rather quickly. Therefore, the distance is set so high. A smaller minimal match length creates more precise results. However, the amount of clones greatly increases and and thus makes it difficult to analyze the results. Additionally, the results reflect similar influences (e.g, a minimal match length of 4 changes the similarity of E to around 96%). Therefore, a minimal match length of 10 is sufficient. The results are presented in Table 4.4.

The expansion obfuscation decreases the size of cross project clones by 79 AST nodes and thus shows an influence on the detection rate. However, most of the affected code fragments are part of a code clone.

Therefore, we can assume that the detector is robust against this obfuscation. However, some code fragments, which are not affected by the obfuscation, are not part of a code clone. It is not identifiable why they are excluded, since there was no change.

The contraction obfuscation decreased the size of crossproject clones by 117. There are no code fragments that are affected by the obfuscation which are part of a code

Obfuscated Project	Total Clones		Cross-Project Clones		
	Clones	Size of Clones	Clones	Size of Clones	Similarity in %
Original	38	2005	22	1793	100
E	41	1926	25	1714	95.6
C	40	1888	24	1676	93.4
EC	43	1809	27	1597	89.0
L	57	1597	41	1385	77.2
EL	58	1565	42	1353	75.5
CL	58	1493	42	1281	71.4
ECL	59	1461	43	1249	69.7
R	53	868	42	730	40.7
RL	54	878	42	730	40.7
RC	53	868	42	730	40.7
RCL	54	878	42	730	40.7
RE	50	828	39	690	38.5
REL	51	838	39	690	38.5
REC	50	828	39	690	38.5
RECL	51	838	39	690	38.5

Table 4.4: Clone Detection Results - AST (Match-length 10 (Tree Nodes))

```
if (item.getType() == GameObject.OBJECT_DOT) {
  dotsRemaining = dotsRemaining + 1;
}
```

```
if (item.getType() == GameObject.OBJECT_DOT) {
  dotsRemaining++;
}
```

Figure 4.4: AST Code Clone - Expansion(with highlighted difference)

clone, thus decreasing the size. Therefore, the clone detector is not robust against a contraction obfuscation. A combination between expansion and contraction combines the two effects mentioned above.

The result of the loop obfuscation shows a decrease of 408 AST nodes. The clone pairs presented in the result file do not contain code fragments with a `for` or `while` loop, thus decreasing the size of crossproject clones. Therefore, the detector is not robust against such transformation. A combination with expansion and contraction combines the effects mentioned above.

Despite normalizing identifiers, the renaming obfuscation has the highest influence on the detection rate. The size of crossproject clones decreases by 1063. However, a huge part of affected fragments are part of code clones.

The decrease of the size of clones is a result of the separation of declaration and initialization of a variable. Due to this separation, a new AST node is created and another node is changed. No affected code fragment is part of a code clone, thus decreasing the size. Combining this obfuscation with contraction and loop has no influence, since

```
public float m104(Actor v41, int v42, int v43, int v44, int v45) {
  return f64.m06(f24, v41, v42, v43, v44, v45);
```

```
public float getHeuristicCost(Actor mover, int x, int y, int tx, int ty) {
  return heuristic.getCost(map, mover, x, y, tx, ty);
```

Figure 4.5: AST Code Clone - Renaming(with highlighted difference)

these parts are already affected by the renaming (e.g, the initializer of a `for` loop). The combination with expansion increases the effect slightly.

In summary, the AST-based clone detection technique is robust against renaming, and expansion, thus matching with our assumption. However, the contraction obfuscations and loop obfuscation weren't found by the detector. The effects of renaming show, that this technique is not robust against reordering or adding/deleting statements.

4.4 Summary

In this thesis, we conducted a case study to determine the robustness of four code clone detection techniques with one representative code clone detector for each technique. We created sixteen different version of the subject system, where each version was obfuscated with a different combination with our obfuscator *ARTIFICE*. Then, each version with the original version was analyzed by the clone detectors. Finally, we analyzed the results to determine the robustness. This let to the results presented in Table 4.5

	Renaming	Expansion	Contraction	Loop Transformation
Textual (JPLAG)	low	low	low	low
Token-based (JPLAG)	high	high	high	low
AST-based (CloneDigger)	high	high	low	low
PDG-based (Scorpio)	medium	low	low	high

Table 4.5: Robustness of Detection Techniques

JPLAG was used for both, textual and token-based. The robustness of textual based detection techniques can be considered as very low. Every obfuscation affected the results and led to lower similarity. The renaming obfuscation is the most effective one, since nearly every line of code is affected.

In contrast to the textual-based technique, the token-based detection technique revealed a high robustness against simple obfuscations. Expansion, contraction as well as renaming showed little to no effect. However, the more complex loop transformation decreased the similarity by nearly 10% with only 34 transformations.

The AST-based technique showed a high robustness against obfuscations, as long no reordering or adding/deletion of statements affected the AST-Tree of the source code. Simple renaming and expansion were detected very well. However, the contraction and

loop transformation decreased the result with every applied obfuscation. Additionally, when initialization and declaration of a variable is split, it is not detected anymore, since a node is added to the tree.

The PDG-based detection technique is robust against more complex obfuscations. The loop transformation had little effect on the detection rate. In combination with other obfuscations, it proofed to beneficial for the detector. The results showed a better detection rate with the loop transformation applied. However, code fragments that were affected by expansion and contraction were not part of a code clone, thus decreasing the similarity. Therefore it is not robust against these obfuscations. *Scorpio* detected code fragment that were affected by renaming. However, the more renamings are applied on one PDG node, the less likely it is part of a code clone.

4.5 Threats to Validity

There are two main topics that can attack the validity of this case study: the implementation of the technique in the specific clone detector, and the source code of the subject system.

The different representation that are used by the clone detectors are not always clearly defined. A token for example, is simply a string of characters. However, it is not defined that a token has to be exactly one line, statement, number, literal or similar. The interpretation of a token is implemented in the software tool and thus dependent on the implementation. *CCFinder* [KKI02] parses one line of code into much more tokens than *JPlag*. Therefore, the results are most likely different. However, the clone detection tools are often cited and known for its reliability. Therefore, we can use them as representation for their respective technique.

The results are highly dependent on the subject system and its source files. The obfuscator is only able to transform as much statements as the source files offer. However, not every system has the same amount of obfuscatable statements. For example, the Pac-Man project contains 8 possible contraction statements and 54 expansion statements. Based on the results of each clone detection process, we can argue that the expansion obfuscation has more impact than the contraction obfuscation. However, with a deeper analysis of the source code and the detection results, we are able to assess certain tendencies regarding each obfuscation. These tendencies are independent from the subject system.

5. Related Work

The survey by Roy et. al [RC07] is a summary of the research on clone detection and discusses the theory of clone cloning, taxonomies, and detection techniques. Additionally, they summarized several experiments including an evalutation of precission and recall of different clone detectors. Furthermore, they conducted an experiment concerning the robustness of code clone techniques, too. Overall, the results resemble the results of this thesis. However, the results give only a short evalutation for each technique and don't offer an insight why several transformations affect the detection rate.

Juergens et al. [JDH10] conducted an experiment to evaluate how well existing clone detection approaches detect similarity in 109 independently developed variations of the same functionality. In other words, the aim of this experiment is to evaluate the performance of detection tool regarding behavioral similar code clones (*Type IV* clones). Their results indicate that current techniques are not capable of finding functional similar code fragments, since the two used clone detectors (ConQAT [JDH09] and DECKARD [JMSG07]) achieved a recall (ratio of detected clone pairs) of less than 1%.

6. Conclusion

In this thesis we conducted an empirical study to evaluate the effect of code obfuscations on the detection rate of different code clone detection tools. Since each tool uses a different clone detection technique, we can make a statement about the robustness of each technique.

Code clones are copied code fragments in the source code of the system, which occur either intentionally or unintentionally. To detect code clones, there are several available software tools. There are four main detection techniques used by the tools - textual, token-based, AST-based, PDG-based. They parse the source code in to the specific representation on which the tools perform the analysis, and then present the code clones in a user-friendly format.

We developed the code obfuscator *ARTIFICE*, to easily apply several code obfuscations on a software system. It is developed as an Eclipse plugin with a simple interface and can perform renaming, expansion, contraction, loop and conditional obfuscations. Furthermore, the respective code transformations are saved in a logfile.

We presented an empirical to evaluate the robustness of each clone detection technique using a representative clone detector. We used a software system consisting of 2415 lines of code as the subject system. Several copies of this system were made, on which we applied different obfuscation combinations using the obfuscator. Each obfuscated copy combined with the original source code was analyzed by each clone detector separately. The results were used to evaluate the robustness of the detection techniques.

The textual-based detection was performed by using the text analysis of *JPLAG*. The results showed a considerably low robustness against every applied obfuscation. Especially the renaming obfuscation decreased the similarity between the original and the obfuscated copy by 41%.

JPLAG was used for the token-based detection, too. The robustness against simple obfuscations that target one line of code can be considered as high. Renaming, ex-

pansion and contraction didn't change the detection results. However, the splitting of declaration and initialization of variables performed by the renaming obfuscation and the loop transformation affected the result. Therefore, the token-based approach is not robust against more complex obfuscations.

The AST-based detection was performed by using *CloneDigger*. As long as the AST of the source code is not affected by the addition/deletion of statements or reordering, the robustness of this technique is high. Code fragments that are changed by expansion or simple renaming are part of a code clone after obfuscation. Therefore, the detector is robust against these obfuscations. However, loop transformation, contraction and splitting declaration and initialization of variables decreased the similarity considerably.

We used *Scorpio* for the PDG-based detection. The results showed that the robustness against loop transformation is high, since effect on the similarity was little. However, code fragments that were affected by expansion and contraction were not part of a code clone after obfuscation. Therefore it is not robust against these obfuscations. *Scorpio* detected code fragment that were affected by renaming. However, the more renamings are applied on one PDG node, the less likely it is part of a code clone.

In summary, we constitute that every clone detection technique(except textual-based) is robust against at least one obfuscation. However, each tool showed weaknesses although we used simple transformations only. We assume that advanced transformations such as reordering or code addition (of dead code) reveal additional vulnerabilities. Furthermore, we showed that there is a danger, that plagiarized software can be made unrecognizable. Therefore, there is the necessity to find new methods and approaches to counter this danger.

7. Future Work

In this thesis, we evaluated the effect of different obfuscations on the detection rate of several clone detectors and gave an insight of the robustness of clone detection techniques. However, there are several topics to evaluate in future works.

The obfuscator *ARTIFICE*, we developed, is able to apply simple obfuscation on source code, where the majority targets single lines of code. We showed the robustness against these obfuscations in our results. However, there are a lot more complex transformation we did not implement. Reordering of statements, inlining and outlining of methods or changing object types are some examples, which highly influence the program and data flow and will result in a much more obfuscated source code. Therefore, the impact on the results of a clone detection process is to be tested.

For each clone detection technique we used one representative clone detector and evaluated its results. With the results, we could assess tendencies towards the robustness against several obfuscations. However, it is possible that other clone detectors have slightly different results. Therefore, it necessary to check whether the results match with the stated tendencies. Furthermore, with more results from each clone detection technique it is possible to give generalized statements regarding the robustness of clone detection tools. Additionally, with an analysis on a larger software system, it is possible to get more precise results, since it will contain more fragments that can be obfuscated.

We analyzed why different obfuscation have an effect on the detection rate of clone detection tools. With the results and the insight how clone detection works, you can work on effective ways to enhance the robustness against these obfuscations, if the clone detection technique allows so. Therefore, it is possible to enhance the effectiveness of clone detection software.

A. Appendix

```
1  \\Expansion
2      [variable]++; --> [variable] = [variable] + 1;
3      [variable]--; --> [variable] = [variable] - 1;
4      \\if variable is of type byte, the righthand side is cast to byte
5
6      [variable] += ...; --> [variable] = [variable] + (...);
7      [variable] -= ...; --> [variable] = [variable] - (...);
8      [variable] *= ...; --> [variable] = [variable] * (...);
9      [variable] /= ...; --> [variable] = [variable] / (...);
10
11 \\Contraction
12     [variable] = ...+ [variable] + ...; --> [variable] += ... + 0 + ...;
13     [variable] = [variable] - ...; --> [variable] -= 0 - ...;
14     \\variable has to be outside brackets
15
16     [variable] = [variable] + 1; --> [variable]++:
17     [variable] = [variable] - 1; --> [variable]--;
18
19 \\Renaming (all references are renamed accordingly)
20     [type] [variable]; --> [type] [newVariable];
21
22     [type] [variable] = [value];
23     --->
24     [type] [newVariable];
25     [newVariable] = [value];
26
27     [type] [field]; --> [type] [newField];
28     [type] [field] = [value]; --> [type] [newField] = [value];
29
30     ...[method]... --> ...[newMethod]...
31
32 \\Loop transformation
33     \\for to while
34     for([initialization]; [expression]; [updater])
```

```
35          [loopStatement];
36      --->
37      [renamedInitialization];
38      while([renamedExpression]) {
39          [LoopStatement];
40          [renamedUpdater];
41      }
42      ____________________________________________________________
43      for([initialization]; [expression]; [updater]) {
44          [loopStatements]
45          }
46      --->
47      [renamedInitialization];
48      while([renamedExpression]) {
49          [LoopStatements];
50          [renamedUpdater];
51      }
52
53      \\while to for
54      while([expression]) {
55          [LoopStatements];
56          [updater];
57      }
58      --->
59      for(;[expression];) {
60          [LoopStatements];
61          [updater];
62      }
63
64  \\Conditional transformation
65      [variable] = [expression] ? [value1]: [value2];
66      <-->
67      if([expression]
68          [variable] = [value1];
69      else
70        [variable] = [value2]
```

Listing A.1: ARTIFICE - Transformations

Bibliography

[AM02] Toshihiro Kamiya Shin-ichi Sato Ken-ichi Matsumoto Akito Monden, Daikai Nakae. Software quality analysis by code clones in industrial legacy software. In *Proceedings of 8th IEEE International Symposium on Software Metrics*, pages 87–94. METRICS 02, June 2002. (cited on Page 3)

[Bak95] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 86–95. WCRE 1995, 1995. (cited on Page 1 and 4)

[BM09] P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *Proceedings of the 3rd International Workshop on Software Clones at CSMR 2009*. Citeseer, 2009. (cited on Page 10)

[CC97] Douglas Low Christian Collberg, Clark Thomborson. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, 1997. (cited on Page vii, 12, 13, and 14)

[HK11] Y. Higo and S. Kusumoto. Code clone detection on specialized pdgs with heuristics. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 75–84. IEEE, 2011. (cited on Page 11, 32, and 35)

[IOC] The International Obfuscated C Code Contest: http://www.ioccc.org/. (cited on Page 15)

[JDH09] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective-a workbench for clone detection research. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 603–606. IEEE, 2009. (cited on Page 39)

[JDH10] E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 78–87. IEEE, 2010. (cited on Page 39)

[JDT] Eclipse - JDT Core Package: http://www.eclipse.org/jdt/core/index.php. (cited on Page 22)

[JM96] Ettore Merlo Jean Mayrand, Claude Leblanc. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the 12th International Conference on Software Maintenance*, pages 244–253. ICSM 96, June 1996. (cited on Page 3)

[JMSG07] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 96–105. IEEE, 2007. (cited on Page 39)

[Joh94] John Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the 10th International Conference on Software Maintenance*, pages 120–126, sept 1994. (cited on Page 3)

[KG06] Cory Kapser and Michael W. Godfrey. Clones considered harmful. In Considered Harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28. WCRE 06, oct 2006. (cited on Page 1 and 3)

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002. (cited on Page 38)

[LP] Michael Phlippsen Lutz Prechelt, Guido Malpohl. JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Department of Computer Science, march. (cited on Page 9)

[RC07] Canchal Kumar Roy and James R. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, School of Computing, sep 2007. (cited on Page 1, 2, 7, and 39)

[RC08] C.K. Roy and J.R. Cordy. An Empirical Study of Function Clones in Open Source Software Systems. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 81–90. WCRE 2008, 2008. (cited on Page 1)

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009. (cited on Page vii, 1, 2, 7, and 8)

[Rey70] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine intelligence*, 5(1):135–151, 1970. (cited on Page 11 and 26)

[SC] Scorpio - PDG-based clone detector: http://sdl.ist.osaka-u.ac.jp/ higo/cgi-bin/moin.cgi/scorpio-e. (cited on Page 32)

[WKCG03] B. Li W-K. Chen and R. Gupta. Code Compaction of Matching Single-Entry Multiple- Exit Regions. In *Proceedings of the 10th Annual International Static Analysis Symposium*, pages 401–417. SAS 03, June 2003. (cited on Page 4)

[WL06] Andrew Walenstein and Arun Lakhotia. The Software Similarity Problem in Malware Analysis. In *Proceedings Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, page 10, July 2006. (cited on Page 4)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 24. Oktober, 2012