Studying the Impact of Obfuscation on Source Code Plagiarism Detection

Albert Cabré Juan

BSc Computer Engineering
January 2014

Under the supervision of Jens Krinke
Department of Computer Science
University College London

Acknowledgments

I would like to thank Prof. Jens Krinke, my supervisor, for his advice, collaboration, patience and continuous guidance during the development of this project. He has played a critical role in this project without which its completion would have been impossible.

I also would like to thank the Department of Computer Science at University College London for offering me the opportunity to finish my degree on it, which has been one of the most fantastic experiences in my life.

Abstract

Software plagiarism detection tools have long existed and been evaluated. However, the impact source code obfuscation has on their performance is a much less studied subject. This project aims to study how different plagiarism detection tools are affected by source code obfuscation and whether there is a way to improve it.

To carry this out, a sample of programs was selected, automatic source code obfuscations were performed on them, and finally the plagiarism detection tools were ran against the originals to evaluate their capacity to assess plagiarism.

As a result, we have detected that source code obfuscation does indeed impact plagiarism detection tools performance, and we present a way to improve it.

Contents

1. Introduction

1.1 Motivation

Although there is some debate regarding the definition of what constitutes software plagiarism, the following definition is considered to be widely accepted (Cosma and Joy, 2008, pp. 195--200): software plagiarism is defined as the act of copying someone else's source code with or without making alterations and submitting it without providing any acknowledgements.

Software plagiarism has classically been a problem in academia environments, where students are asked to work in the same problem within a common environment. This has long been perceived as an increasing problem, with surveys showing that more than half of the academic staff think software plagiarism is an increasing problem (Bull and Collins et al., 2001).

However, recent events such as the Google vs Oracle case, shows that software plagiarism might be a much more widespread problem than it was previously thought. Even though the jury didn't find Google guilty of infringing Oracle's patents, it was clear that Google had plagiarised Oracle's source code, perhaps in a legal way (Fiducia and Briskman, 2013).

Software plagiarism is usually prevented through the use of automatic tools that attempt to detect it. However, a lot of the most popular tools weren't originally developed with this specific problem in mind, but to detect either software clones or non-software plagiarism. In the case of software plagiarism, it is very common for plagiarists to try to disguise their plagiarism through some kind of source code obfuscation, a scenario for which most of this tools were never prepared.

Consequently, this project will aim to evaluate whether current software plagiarism detection tools are effective or not in the presence of source code obfuscation. To achieve this, a large scale evaluation will be done testing the most popular tools and approaches against a wide range of obfuscation techniques and environments.

The main goal is to find what are the best tools and settings to detect software plagiarism in the presence of source code obfuscation, and if no satisfying tools are found, provide directions for the development of future tools.

1.2 Project development

This project was developed under an iterative approach, with weekly tutor meetings. First, different tools were collected and tested manually, keeping the ones that were providing encouraging results, and disregarding the ones that didn't. Because we planned to run these later in an automatic way, some custom scripts had to be written to make some of the tools work the way we wanted to at this stage.

We also decided which test source code we were going to use. We chose a variety of classic algorithmic problems that almost every computer science student has to implement at some point in his degree. Some source codes contained pieces of code that were problematic for a lot of tools (like inner-classes, which several obfuscators can't handle well), and so were disregarded later.

Later, we chose the obfuscation tools. We wanted to apply a variety of obfuscation techniques to all the test data, and so we chose some of the cutting-edge source code obfuscators that work directly on the bytecode coupled with a variety of decompilers. We also used artifice, a tool that simulates the obfuscations plagiarists usually apply manually directly on the source code (Schulze and Meyer, 2013, pp. 62 - 68).

Finally we wrote scripts that automatically applied the obfuscations, that decompiled the obfuscated bytecode, and that finally ran the plagiarism detection tools on all the generated sources. We also developed scripts that tried a wide range of parameters for the plagiarism detection tools that allowed them, to find the best settings for each detector.

1.3 Chapter overview

Chapter two gives information on the context of this project, that is, information necessary to understand the project but that is not specific to this project. A short explanation of previous existing related work is given, and also information on the tools and programs that will be used.

Chapter three explains everything related to the work that was done before actually running the plagiarism detection tools, that is mainly running the obfuscators on the test programs and analysing the results.

Chapter four explains the process of running the plagiarism detection tools, presents the results and analyses them. Finally, chapter five explains the conclusions of the project.

2. Context

2.1 Background

Several studies have been made regarding source code plagiarism detectors, but almost none of them deal with source code obfuscation. Of special interest is another study (Roy and Cordy et al., 2009, pp. 470--495), where a large scale comparison between different plagiarism detectors tools and techniques is performed (but again, without taking source code obfuscation into account). However, this study was very useful in providing information regarding the different tools and techniques and inspiration as to how to perform an evaluation of this kind.

Very recently, a study on plagiarism detection in presence of source code obfuscation was performed (Schulze and Meyer, 2013, pp. 62 - 68) , this being the main resource this project has been built upon. This paper was based on another student bachelor's thesis, in which he developed ARTIFICE, an Eclipse plugin that performs transformations on Java source code that resemble the kind of transformations plagiarists usually perform on source code. We used this tool as an obfuscator in our study. Aside from the development of this tool, this thesis also performed a study on plagiarism detection in the presence of source code obfuscation. However, this study was limited to three plagiarism detectors and ARTIFICE as the only obfuscator. As a consequence, this project started by replicating their results initially and then greatly expanding it from there.

2.2 Test data set

Java was chosen as the target language for our study from the very beginning for various reasons: it's one of the most extended languages both in academia and in enterprises, it's object oriented (which encourages reuse of code), and a lot of both obfuscation and plagiarism detection tools exist for it. We chose some well-known algorithms implementations in Java:

- InfixConverter: This program takes an Infix notated arithmetic formula and returns the postfix notated equivalent. This problem is common on data structures courses, as it demonstrates the usages of a stack structure.
- SqrtAlgorithm: This program calculates the square root of an integer without using the built-in sqrt() function. It is interesting because it has a high density of

flow control structures.

- Hanoi: This program solves the Towers of Hanoi problem in the minimum steps possible with an arbitrary number of disks. It is a classic computer science problem that is very commonly used to demonstrate recursion in advanced programming courses.
- Queens: This program solves the 8 queens puzzle but with an arbitrary number of queens using recursion. As with the Hanoi program, it’s a common problem for advanced programming courses students.
- MagicSquare: This program finds all the correct Magic Squares of an arbitrary size. It is a common problem used to demonstrate backtracking in programming courses.

2.3 Source code obfuscators

Different obfuscators were applied to all the test data, producing its respective obfuscated versions for later testing with the plagiarism detectors.

2.3.1 ARTIFICE

ARTIFICE performs transformations directly on the source code like renaming variables, transforming loops structure (for loops are transformed into while loops and vice-versa), expanding or contraction of variable definitions (the variable declaration and its value assignment are separated into multiple lines or merged into a single one), and conditional statements transformations (if/else statements are transformed into a ternary operator statement and vice-versa).

2.3.2 ProGuard

ProGuard is a free Java class file shrinker, optimizer, obfuscator, and preverifier. It detects and removes unused classes, fields, methods, and attributes. It optimizes bytecode and removes unused instructions. It renames the remaining classes, fields, and methods using short meaningless names” (Lafortune, 2013).

ProGuard has been getting a lot of attention lately as it became part of the default Android build system (AOSP, 2013). We decided to use ProGuard as it is the most used Java obfuscator nowadays. However, since it works directly with the bytecode, we had to compile and decompile too. This introduces more obfuscation, as explained below.

2.4 Plagiarism detectors

2.4.1 Jplag

Jplag is a token based plagiarism detector. It works by first creating a token string for each program to be compared, then it compares the two token strings by trying to recreate one of the strings with substrings of another token string. How much of the string can be recreated is the similarity measure outputted. This algorithm is known as Greedy String Tiling (Wise, 1993).

Jplag supports multiple languages, and the difference between running Jplag in the different modes for different languages is in how the token strings are created. If running in text mode, the tokens will be parts of the text itself, but when running in Java mode, for instance, the tool will understand the internal structure of the code and transform the code into tokens like “vardef” or “beginmethod” (Prechelt and Malpohl et al., 2002, p. 1016).

Although we are using Java, we will run Jplag both in Java and text mode, as the comparison of the results produced by the two different methods of generating the token string could yield interesting data. We will refer to these 2 by “Jplag-text” and “Jplag-java”.

2.4.2 CCFinder

CCFinder works in a very similar way as JPlag, as it is token based too, with two main differences. First, the algorithm to create the token strings understands the source code internal structure in much greater detail. As a result, CCFinder produces a lot more tokens for the same source code than JPlag (Meyer and Schulze, 2012). Secondly, instead of the Greedy String Tiling algorithm, CCFinder uses a suffix tree matching algorithm (Kamiya and Kusumoto et al., 2002, pp. 654--670). This two differences make the two tools produce different results and because both of them are very popular, they both have been included in this study.

2.4.3 NICAD (Near-miss Intentional Clone Automatic Detection)

NICAD is one of the newest tools. It first parses the code using TXL, a special programming language for parsing other programming languages source code. This allows NICAD to support any programming language source code as long as the TXL grammar for that language is provided. Using TXL, NICAD extracts the fragments of a given granularity, which can be set as a parameter, for instance functions. Once it has all the functions they are normalised by removing whitespace and similar transformations, and then each function in this case is compared line by line using a longest common subsequence algorithm (Cordy and Roy, 2011, pp. 219--220). The most interesting part is that unlike in other approaches, once the source code has been parsed into fragments it's treated as regular text for the comparison. This hybrid approach is what makes NICAD a unique plagiarism detection tool.

2.4.4 Plaggie

Plaggie another token based detector that shares a lot with Jplag. They both first tokenize the source code and use the Greedy String Tiling algorithm to compare the tokens (Hage and Rademaker et al., 2010, p. 28). However, Jplag is closed source and must be run online on their servers, whereas Plaggie is an open source implementation that runs locally. Because the approach that Jplag uses is fairly popular amongst plagiarism detectors, we thought it was sensible to include a popular alternative too.

2.4.5 Sherlock

Sherlock is yet another token based plagiarism detector. As such, it first converts the source code into tokens (they call these digital signatures) which are later used for comparison. Sherlock is open source and very lightweight, the whole code consisting of only slightly over 300 lines of C code. Sherlock is the plagiarism detector used by BOSS (Warwick, 2009), the platform University of Warwick uses to process all the student submissions, and it has been so for over 10 years.

2.4.6 SIM

SIM has been around for 25 years now, and while it's not developed nor maintained anymore it's still actively used to check students' submissions at the VU University of Amsterdam. It is a token based detector, and in fact, it was one of the firsts plagiarism detection tools to use this approach. SIM then compares the two token strings and reports their similarity as the percentage of the first token string that can be constructed using substrings of the second token string of minimum length N, N being the main parameter that has to be set for the comparison (Grune and Vakgroep, 1989). SIM supports textual text or a variety of programming languages, the difference being how the tokens translation is done. We will use both the textual text (simtext) and the java (simjava) modes.

2.4.7 NCD (Normalized Compression Distance)

The Normalized Compression Distance between two objects is a metric that compares the sizes of the compressed version between two objects with the size of the result of compressing both objects together. Because compression algorithms take advantage of repeated parts of the file to create an as small as possible compressed file that contains all the entropy of the original file, compressing two plagiarised files together will produce a smaller sized file than the sum of both compressed files. The larger this difference is, the more similar the two files are. The NCD has been used to detect plagiarism before (Cebrian and Alfonseca et al., 2009, pp. 477--485). Because the NCD is dependent on which compression algorithm is used, three of the most popular compressors will be used for this: Zlib, Gzip and bzip2.

2.4.8 Cosine Similarity

If we take a document, and create a multi-space vector, where each term in the document is a different dimension, and its value is the frequency of the term in the document, we obtain the term vector model of the document (Salton and Wong et al., 1975, pp. 613--620). If we then calculate the cosine similarity between the vectors obtained from two documents, we obtain a metric that is valuable in measuring how similar the two documents are. This technique is used already in some plagiarism detection tools (Si and Leong et al., 1997, pp. 70--77).

We have built a tool that using Apache’s Lucene, one of the most popular text analysis libraries, obtains the term vectors for different documents and then outputs the cosine similarity between them. For redundancy reasons, we will also compute this metric using SkLearn, a machine learning library for python which allows to obtain the vector model of the documents similarly.

2.4.9 Gestalt Pattern Matching (DiffLib)

The Gestalt Pattern Matching algorithm attempts to output a metric that represents the similarity between two strings without using any kind of preprocessing or tokenization (W. Ratcliff and E. Metzener, 1982, pp. 46--51). While it was originally not conceived as a plagiarism detection algorithm, it has been applied to clone detection (Kontogiannis and Demori et al., 1996, pp. 77--108), and so is interesting to us as well.

We’re using a python implementation based on this algorithm, DiffLib, although they have tuned and improved the original algorithm.

2.4.10 Jaro Distance (jellyfish)

The Jaro Distance, and the latter improved version, the Jaro-Winkler Distance, also analyse two strings directly to output a similarity metric. While they were conceived for short string such as names (Cohen and Ravikumar et al., 2003, pp. 73--78), it has been used to detect plagiarism (Rahman, and Puspitodjati, 2009).

We will use a python implementation of both versions of the algorithm, which we can find under the jellyfish python library.

2.4.11 N-gram similarity (NGram)

The N-gram similarity technique consists of converting the documents to sets of n-grams, and then comparing those sets to obtain the n-gram similarity (Kondrak, 2005, pp. 115--126). This technique has been applied to plagiarism detection before (Barrón-Cedeño and Rosso, 2009, pp. 696--700) and so is of interest to us.

We will use the python library NGram, which provides string similarity functionality.

2.4.12 Sort Unique

Sorting by unique lines is a primitive compression algorithm which actually only removes duplicate lines. As with other compression related techniques, by comparing the size of two individually compressed files with the size of the result of compressing them both together we can obtain a metric of how much duplicated content was there in the files.

2.5 Compilers

2.5.1 Javac

Because it’s the compiler used almost universally, it’s the only compiler we will use.

Simply compiling Java source code removes a lot of information (the compiler introduces casts to deal with generics for instance). It is for this reason that the compilation is considered an obfuscation transformation in itself. This need to be taken into account when analyzing the results.

2.6 Decompilers

2.6.1 Decompilation (Procyon)

Procyon is one of the most popular open source Java decompilers and so we chose it as our standard decompiler.

2.6.2 Decompilation (Krakatau)

Since decompilers have to do a lot of guessing work, the guessing strategies vary from one decompiler to the other. It is for this reason that aside from the standard decompiler we wanted to use something else too. Krakatau is a rather different decompiler.

“The Krakatau decompiler takes a different approach to most Java decompilers. It can be thought of more as a compiler whose input language is Java bytecode and whose target language happens to be Java source code. Krakatau takes in arbitrary bytecode, and attempts to transform it to equivalent Java code. This makes it robust to minor obfuscation, though it has the drawback of not reconstructing the "original" source, leading to less readable output than a pattern matching decompiler would produce for unobfuscated Java classes.” (Grosse, 2013).

The key concept here is that Krakatau uses a much more heuristic approach to the decompilation, and as a result the transformation is supposed to be much stronger, which is very interesting to us.

3. Design & Implementation

3.1 Test data preparation

As explained earlier in 2.1, we have a series of Java programs that we intend to use as our test data. The first step towards analysing the effectiveness of plagiarism detection tools is obviously to produce obfuscate the code. As mentioned before, we'll be focusing in two obfuscators: artifice and ProGuard, so the first step is to apply those to each of our programs.

Artifice works directly with the source code, so we just applied it manually to each of the programs. ProGuard, however, works at the bytecode level, and so we will have to first compile our programs, apply ProGuard and then decompile them. Because the decompilation phase might produce different results depending on the decompiler, we already said we were going to use two different decompilers.

Also, because compiling and decompiling produces a source code that is functionally equivalent to the original but where the code is different, we can consider this to be an obfuscation in itself.

Last but not least, we cannot ignore the possibility of using more than one obfuscation at the same time, so we want to also study combinations of all the listed above.

So, at this point, for each program (0_orig), we produce 9 versions of it:

- An artifice obfuscated version (1_artifice).
- A version that has been compiled and decompiled with Krakatau (test_0_orig_no_krakatau).
- A version that has been compiled and decompiled with Procyon (test_0_orig_no_procyon).
- A version that has been compiled, obfuscated with ProGuard, and decompiled with Krakatau (test_0_orig_pg_krakatau).
- A version that has been compiled, obfuscated with ProGuard, and decompiled with Procyon (test_0_orig_pg_procyon).
- A version that has been obfuscated with artifice, compiled, and decompiled with Krakatau (test_1_artifice_no_krakatau).
- A version that has been obfuscated with artifice, compiled, and decompiled with Procyon (test_1_artifice_no_procyon).
- A version that has been obfuscated with artifice, compiled, obfuscated with

ProGuard, and decompiled with Krakatau (test_1_artifice_pg_krakatau).
- A version that has been obfuscated with artifice, compiled, obfuscated with ProGuard, and decompiled with Procyon (test_1_artifice_pg_procyon).

Everything except the artifice obfuscation (which works as an Eclipse plugin) is automated for easy expandability.
3.2 The obfuscated test data

Following is an example from each of these versions, concretely, the main method of the Hanoi Towers program.

3.2.1 0_orig:

```
public static void main ( String[] args ) {
    try {
        while ( true ) {
            System.out.print ( "\nEnter the number of discs (-1 to exit): " );
            int maxdisc = 0;
            String inpstring = "";
            InputStreamReader input = new InputStreamReader ( System.in );
            BufferedReader reader = new BufferedReader ( input );
            inpstring = reader.readLine();
            movecount = 0;
            maxdisc = Integer.parseInt ( inpstring );
            if ( maxdisc == -1 ) {
                System.out.println ( "Good Bye!" );
                return;
            }
            if ( maxdisc <= 1 || maxdisc >= 10 ) {
                System.out.println ( "Enter between 2 - 9" );
                continue;
            }
            for ( int i = maxdisc; i >= 1; i-- ) {
                A.push ( i );
            }
            countA = A.size();
            countB = B.size();
            countC = C.size();
            PrintStacks();
            SolveTOH ( maxdisc, A, B, C );
            System.out.println ( "Total Moves = " + movecount );
            while ( C.size() > 0 ) {
                C.pop();
            }
        }
    } catch ( Exception e ) {
        e.printStackTrace();
    }
}
```

3.2.2 1_artifice:

```
public static void main ( String[] v11 ) {
  try {
    for ( ; true; ) {
      System.out.print ( "\nEnter the number of discs (-1 to exit): " );
      int v12;
      v12 = 0;
      String v13;
      v13 = "";
      InputStreamReader v14;
      v14 = new InputStreamReader ( System.in );
      BufferedReader v15;
      v15 = new BufferedReader ( v14 );
      v13 = v15.readLine();
      f00 = 0;
      v12 = Integer.parseInt ( v13 );
      if ( v12 == -1 ) {
        System.out.println ( "Good Bye!" );
        return;
      }
      if ( v12 <= 1 || v12 >= 10 ) {
        System.out.println ( "Enter between 2 - 9" );
        continue;
      }
      int v16;
      v16 = v12;
      while ( v16 >= 1 ) {
        f10.push ( v16 );
        v16 = v16 - 1;
      }
      f40 = f10.size();
      f50 = f20.size();
      f60 = f30.size();
      m20();
      m10 ( v12, f10, f20, f30 );
      System.out.println ( "Total Moves = " + f00 );
      for ( ; f30.size() > 0; ) {
        f30.pop();
      }
    }
  } catch ( Exception v17 ) {
    v17.printStackTrace();
  }
}
```

3.2.3 test_0_orig_no_krakatau

```
public static void main ( String[] a ) {
        try {
            while ( true ) {
                System.out.print ( "\nEnter the number of discs (-1 to exit): " );
                String s = new java.io.BufferedReader ( ( java.io.Reader ) new java.io.InputStreamReader (
System.in ) ).readLine();
                movecount = 0;
                int i = Integer.parseInt ( s );
                if ( i != -1 ) {
                    if ( i > 1 && i < 10 ) {
                        int i0 = i;
                        while ( i0 >= 1 ) {
                            A.push ( ( Object ) Integer.valueOf ( i0 ) );
                            i0 = i0 + -1;
                        }
                        countA = A.size();
                        countB = B.size();
                        countC = C.size();
                        Main.PrintStacks();
                        Main.SolveTOH ( i, A, B, C );
                        System.out.println ( new StringBuilder().append ( "Total Moves = " ).append ( movecount
).toString() );
                        while ( C.size() > 0 ) {
                            C.pop();
                        }
                        continue;
                    }
                    System.out.println ( "Enter between 2 - 9" );
                } else {
                    break;
                }
            }
            System.out.println ( "Good Bye!" );
        } catch ( Exception a0 ) {
            a0.printStackTrace();
            return;
        }
    }
```

3.2.4 test_0_orig_no_procyon

```
    public static void main ( final String[] array ) {
        try {
            while ( true ) {
                System.out.print ( "\nEnter the number of discs (-1 to exit): " );
                final String line = new BufferedReader ( new InputStreamReader ( System.in ) ).readLine();
                Main.movecount = 0;
                final int int1 = Integer.parseInt ( line );
                if ( int1 == -1 ) {
                    break;
                }
                if ( int1 <= 1 || int1 >= 10 ) {
                    System.out.println ( "Enter between 2 - 9" );
                } else {
                    for ( int i = int1; i >= 1; --i ) {
                        Main.A.push ( i );
                    }
                    Main.countA = Main.A.size();
                    Main.countB = Main.B.size();
                    Main.countC = Main.C.size();
                    PrintStacks();
                    SolveTOH ( int1, Main.A, Main.B, Main.C );
                    System.out.println ( "Total Moves = " + Main.movecount );
                    while ( Main.C.size() > 0 ) {
                        Main.C.pop();
                    }
                }
            }
            System.out.println ( "Good Bye!" );
        } catch ( Exception ex ) {
            ex.printStackTrace();
        }
    }
```

3.2.5 test_0_orig_pg_krakatau

```
    public static void main ( String[] a0 ) {
        try {
            while ( true ) {
                System.out.print ( "\nEnter the number of discs (-1 to exit): " );
                String s = new java.io.BufferedReader ( ( java.io.Reader ) new java.io.InputStreamReader (
System.in ) ).readLine();
                a = 0;
                int i = Integer.parseInt ( s );
                if ( i != -1 ) {
                    if ( i > 1 && i < 10 ) {
                        int i0 = i;
                        while ( i0 > 0 ) {
                            b.push ( ( Object ) Integer.valueOf ( i0 ) );
                            i0 = i0 + -1;
                        }
                        e = b.size();
                        f = c.size();
                        g = d.size();
                        Main.a();
                        Main.a ( i, b, c, d );
                        System.out.println ( new StringBuilder ( "Total Moves = " ).append ( a ).toString() );
                        while ( d.size() > 0 ) {
                            d.pop();
                        }
                        continue;
                    }
                    System.out.println ( "Enter between 2 - 9" );
                } else {
                    break;
                }
            }
            System.out.println ( "Good Bye!" );
        } catch ( Exception a1 ) {
            a1.printStackTrace();
            return;
        }
    }
```

3.2.6 test_0_orig_pg_procyon

```
public static void main ( final String[] array ) {
    try {
        while ( true ) {
            System.out.print ( "\nEnter the number of discs (-1 to exit): " );
            final String line = new BufferedReader ( new InputStreamReader ( System.in ) ).readLine();
            Main.a = 0;
            final int int1;
            if ( ( int1 = Integer.parseInt ( line ) ) == -1 ) {
                break;
            }
            if ( int1 <= 1 || int1 >= 10 ) {
                System.out.println ( "Enter between 2 - 9" );
            } else {
                for ( int i = int1; i > 0; --i ) {
                    Main.b.push ( i );
                }
                Main.e = Main.b.size();
                Main.f = Main.c.size();
                Main.g = Main.d.size();
                a();
                a ( int1, Main.b, Main.c, Main.d );
                System.out.println ( "Total Moves = " + Main.a );
                while ( Main.d.size() > 0 ) {
                    Main.d.pop();
                }
            }
        }
        System.out.println ( "Good Bye!" );
    } catch ( Exception ex ) {
        ex.printStackTrace();
    }
}
```

3.2.7 test_0_artifice_no_krakatau

```
public static void main ( String[] a ) {
    try {
        while ( true ) {
            System.out.print ( "\nEnter the number of discs (-1 to exit): " );
            String s = new java.io.BufferedReader ( ( java.io.Reader ) new java.io.InputStreamReader (
System.in ) ).readLine();
            f00 = 0;
            int i = Integer.parseInt ( s );
            if ( i != -1 ) {
                if ( i > 1 && i < 10 ) {
                    int i0 = i;
                    while ( i0 >= 1 ) {
                        f10.push ( ( Object ) Integer.valueOf ( i0 ) );
                        i0 = i0 - 1;
                    }
                    f40 = f10.size();
                    f50 = f20.size();
                    f60 = f30.size();
                    Main.m20();
                    Main.m10 ( i, f10, f20, f30 );
                    System.out.println ( new StringBuilder().append ( "Total Moves = " ).append ( f00
).toString() );
                    while ( f30.size() > 0 ) {
                        f30.pop();
                    }
                    continue;
                }
                System.out.println ( "Enter between 2 - 9" );
            } else {
                break;
            }
        }
        System.out.println ( "Good Bye!" );
    } catch ( Exception a0 ) {
        a0.printStackTrace();
        return;
    }
}
```

3.2.8 test_0_artifice_no_procyon

```
public static void main ( final String[] array ) {
    try {
        while ( true ) {
            System.out.print ( "\nEnter the number of discs (-1 to exit): " );
            final String line = new BufferedReader ( new InputStreamReader ( System.in ) ).readLine();
            Main.f00 = 0;
            final int int1 = Integer.parseInt ( line );
            if ( int1 == -1 ) {
                break;
            }
            if ( int1 <= 1 || int1 >= 10 ) {
                System.out.println ( "Enter between 2 - 9" );
            } else {
                for ( int i = int1; i >= 1; --i ) {
                    Main.f10.push ( i );
                }
                Main.f40 = Main.f10.size();
                Main.f50 = Main.f20.size();
                Main.f60 = Main.f30.size();
                m20();
                m10 ( int1, Main.f10, Main.f20, Main.f30 );
                System.out.println ( "Total Moves = " + Main.f00 );
                while ( Main.f30.size() > 0 ) {
                    Main.f30.pop();
                }
            }
        }
        System.out.println ( "Good Bye!" );
    } catch ( Exception ex ) {
        ex.printStackTrace();
    }
}
```

3.2.9 test_0_artifice_pg_krakatau

```
public static void main ( String[] a0 ) {
    try {
        while ( true ) {
                System.out.print ( "\nEnter the number of discs (-1 to exit): " );
                String s = new java.io.BufferedReader ( ( java.io.Reader ) new java.io.InputStreamReader (
System.in ) ).readLine();
                a = 0;
                int i = Integer.parseInt ( s );
                if ( i != -1 ) {
                    if ( i > 1 && i < 10 ) {
                        int i0 = i;
                        while ( i0 > 0 ) {
                            b.push ( ( Object ) Integer.valueOf ( i0 ) );
                            i0 = i0 + -1;
                        }
                        e = b.size();
                        f = c.size();
                        g = d.size();
                        Main.a();
                        Main.a ( i, b, c, d );
                        System.out.println ( new StringBuilder ( "Total Moves = " ).append ( a ).toString() );
                        while ( d.size() > 0 ) {
                            d.pop();
                        }
                        continue;
                    }
                    System.out.println ( "Enter between 2 - 9" );
                } else {
                    break;
                }
            }
            System.out.println ( "Good Bye!" );
        } catch ( Exception a1 ) {
            a1.printStackTrace();
            return;
        }
    }
```

3.2.10 test_0_artifice_pg_procyon

```
    public static void main ( final String[] array ) {
        try {
            while ( true ) {
                System.out.print ( "\nEnter the number of discs (-1 to exit): " );
                final String line = new BufferedReader ( new InputStreamReader ( System.in ) ).readLine();
                Main.a = 0;
                final int int1;
                if ( ( int1 = Integer.parseInt ( line ) ) == -1 ) {
                    break;
                }
                if ( int1 <= 1 || int1 >= 10 ) {
                    System.out.println ( "Enter between 2 - 9" );
                } else {
                    for ( int i = int1; i > 0; --i ) {
                        Main.b.push ( i );
                    }
                    Main.e = Main.b.size();
                    Main.f = Main.c.size();
                    Main.g = Main.d.size();
                    a();
                    a ( int1, Main.b, Main.c, Main.d );
                    System.out.println ( "Total Moves = " + Main.a );
                    while ( Main.d.size() > 0 ) {
                        Main.d.pop();
                    }
                }
            }
            System.out.println ( "Good Bye!" );
        } catch ( Exception ex ) {
            ex.printStackTrace();
        }
    }
```

3.3 Obfuscation results analysis

Having obfuscated all the test data, we can inspect the results to see how each obfuscation method performed. Most of the observations explained here can be observed on the example main method show in 3.2.

3.3.1 Artifice

Artifice acts in a very predictable way. All of the follow transformations are applied to the source code:

- All loops are transformed. While loops are transformed into for loops, and for loops are transformed into while loops.
- All variables are renamed following a numeric scheme: v1, v2, v3…
- The increment (++), decrement (--) and complex assignment operators (+=, -=, *=, /=) are eliminated but their logic is reconstructed (i++ becomes i = i+1, i*=5 becomes i = i*5...)
- When the operators mentioned in the point above are missing but their logic exists, the reverse process happens, where the statement is simplified

introducing those operators.

3.3.2 ProGuard

- The scope of variables and methods is changed to the minimum possible. That is, all public methods or methods with default visibility are changed to private when possible.
- Variable and function names are changed to letters (a, b, c…)
- Empty strings are injected in string building lines, such as “”+a+b instead of just a + b.
- Variable assignments on declaration are split in two lines: declaration and then value assignment.
- Boolean expressions involving numeric comparisons against constants are changed. For instance, i>=1 becomes i>0.
- Unused variables are removed.

3.3.3 Procyon

- Unused imports are removed.
- Modifiers order is standardized (public before final, etc.)
- Variable declarations are all moved to the top of the code
- Unary operators can get changed from a++ to ++a
- Variables whose value doesn’t change during the method are changed to final, even the method parameters.
- Variable names is changed to what Procyon thinks is going to be more readable. Integers get names like “n”, but variables of type Stack get names like stack1, stack2…
- Similarly to artifice, unary operators are introduced when possible.
- Similarly to artifice, variable declarations are separated from their value assignment. If these were static variables, their value assignment is done separately in a static{} block.
- Variable scope is explicitly declared. A Class variable will get referenced as Main.x, even if there’s no other x.
- If multiple lines can be merged into a single line, they might get merged even when that implies removing some temporary variables that were being used to pass values between the lines.
- Return statements on void methods get changed to breaks if all the code is in an enclosing block.
- Lines that are only executed before a break/return statement are moved outside

the block.

3.3.4 Krakatau

Krakatau does most of the things Procyon does with some core differences:

- Unlike Procyon, unary operators are not preferred, and they get converted to their normal equivalents.
- Imports are removed altogether, with explicit references each time something is used: java.util.Stack().
- When an object is passed to a function that specifies a superclass of the object as the parameter type, a redundant cast to the superclass is added.
- Complex redundant control flow aggregations can get changed. For instance, in the example we can see how a loop that as an if/else block inside where the if has a continue statement gets changed to a single if with the inverse condition and the lines that were inside the if before are moved outside the conditional block.

3.4 Helping plagiarism detectors?

One additional question is whether the performance of plagiarism detection tools in presence of source code obfuscation can be enhanced in some way. Ideally, we would want a deobfuscator, that is, a software that reverts the changes an obfuscator might have introduced, however this is impossible since most of the times, this kind of obfuscations are done manually.

We have observed how the process of compiling and decompiling with either Procyon or Krakatau produces serious changes on the source code, and how some of those changes are similar in nature to those used by obfuscators. In fact, some of them are the exact opposite of what some obfuscators did, like converting all unary operators or complex assignation operators.

For this reason, we will produce two additional data sets, each from running a compilation and decompilation with Krakatau and Procyon respectively, on all the test programs after they have been obfuscated by the means explained earlier. Then we will run the exact same analysis on those, to see if this has helped the plagiarism detection tools at all.

3.5 Experiment design

Once we have generated the 9 different obfuscated versions of each program, we are ready to start running plagiarism detection tools on them. All the plagiarism detection tools work by assigning a score (and for the ones that don't, we have created a metric that is calculated out of the output) of how similar two programs are, not by returning a boolean answer on whether a program has been plagiarised from another program, which is what we want to know ultimately.

It is for this reason that we need to find out what is the best threshold value. The threshold value is the value for which scores under it are considered non-plagiarised and scores equal or above it are considered plagiarised. For any threshold value we can build its confusion matrix (Hamilton, 2012). Different threshold values produce more or less false positives (FP), false negatives (FN), true positives (TP) and true negatives (TN), so we have to choose what we want to maximize or minimize:

- Accuracy (AC): The accuracy is the proportion of detections that were correct, and is defined as the sum of true positives and true negatives divided by the total number of results:

$$AC = \frac{TP \times TN}{TP + TN + FP + FN}$$

- Sensitivity or True Positive Rate (TPR): The proportion of true plagiarism instances that were correctly identified:

$$TPR = \frac{TP}{TP + FN}$$

- Fall-out or False Positive Rate (FPR): The proportion of non-plagiarism instances that were misidentified as plagiarism:

$$FPR = \frac{FP}{FP + TN}$$

A low fall-out rate means the plagiarism detection tool almost never identifies something as plagiarism if it's not. This is very critical in scenarios where the amount of programs to analyse is huge and human intervention is needed for every positive. If the fallout rate is 1% and a million analysis are performed, where none is plagiarised, 10.000 instances will still be identified as plagiarism!

On the other hand, a high sensitivity is critical in scenarios where the highest priority is that the number of plagiarism instances that go undetected is minimal.

Ideally, we would like to have a 0% fall-out rate and a 100% sensitivity, which in turn, means a 100% Accuracy. However, this almost never possible and a compromise has to be made in between depending on the context. Because this project wants to cover both scenarios, we will treat False Positives and False Negatives equally, so we will be trying to maximise the Accuracy.

4. Experiment and Evaluation

Having all the test data prepared, the only thing that’s left is to run the plagiarism detection tools on them and analyse the results.

4.1 Plagiarism detection tools parameters

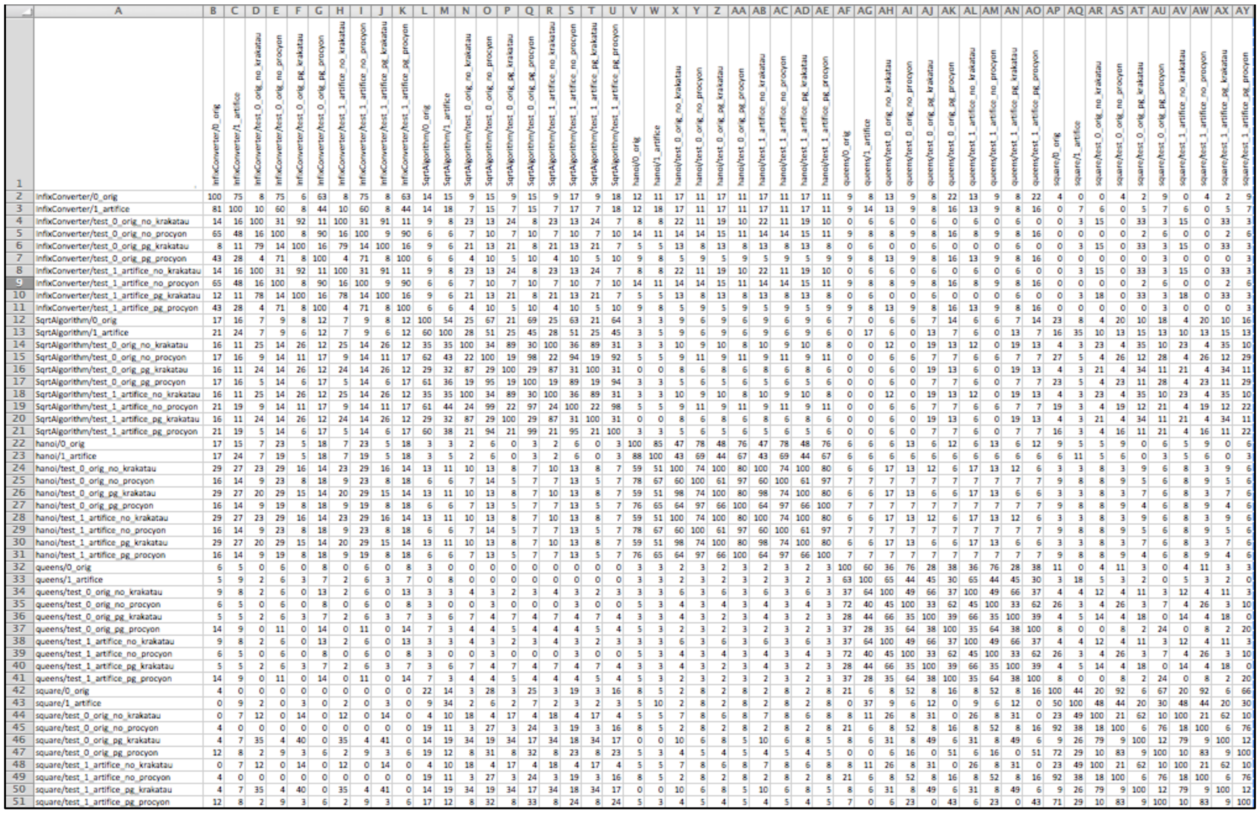
A very important step to run the plagiarism detection tools is to select the parameters we want to tweak so that we can run them with different values for those in order to obtain the absolute best performance the tool can provide. Next is a short explanation for the parameters that we are considering in each tool. The ranges in which the parameters are tested is based on previous testing and on the default values for them, which is usually the central value of the range.

- Bzlib, Zlib, the Lucene based custom tool, the sort unique technique and all python libraries take no parameters.
- Bzip2 allows the minimum compression block size to be set from 100kb to 900kb. This is what ultimately makes it compress softer or harder. The problem is that even the biggest of our programs source code is smaller in size than the minimum block size, so even though all possible sizes were tried, they produced no changes in the result.
- With Gzip we are setting the compression level from 1 to 9.
- With Ccfinder we are setting the minimum number of tokens that can constitute a clone from 0 to 100, and the minimum of kinds of tokens that have to appear in a sequence for it to be able to be considered a clone from 9 to 1.
- With Jplag we are setting the minimum number of tokens that can constitute a clone from 1 to 12.
- With NiCad we are setting its internal threshold parameter (not to be confused with the threshold used to build our confusion matrix) from 40 to a 100, which specifies which percentage of a clone can be different than its original while still being detected as a clone.
- With Plaggie, we are setting the minimum match length from 14 to 1, which is again the minimum number of consecutive tokens that can constitute a clone.
- With Sherlock, we are setting the number of words that are used to form a digital signature from 8 to 1. We are also setting the granularity of the comparison, which can go from 0 (only exactly equal digital signatures are counted) to 8 (more room for differences is given).
- With SIM we’re setting the minimum run length from 28 to 19 when running in

java mode and from 12 to 4 when running in text mode.

4.2 Running the plagiarism detection tools

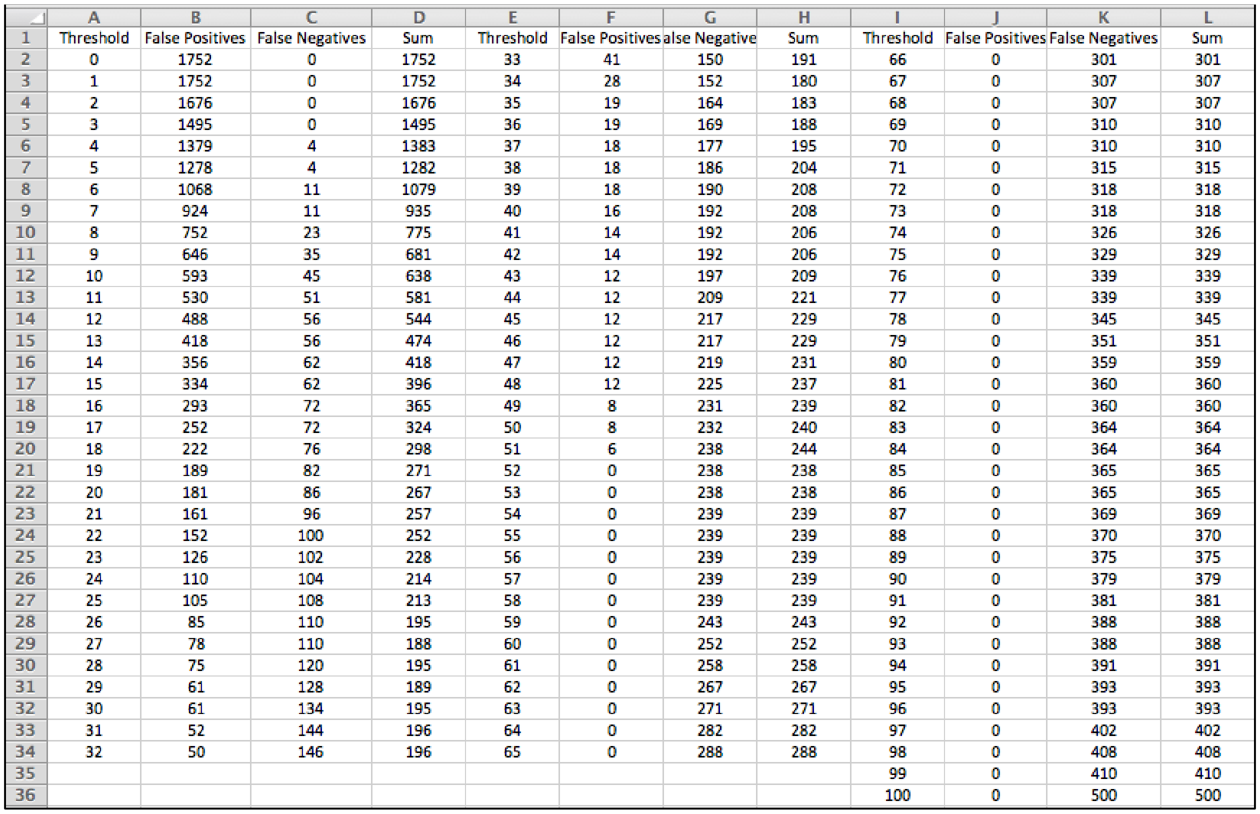
As explained earlier, we want to run each tool on all the test programs pairwise. For this, we’ve created specific scripts for each tool that give us a standardized output. This consists of a CSV file that contains a table with all the pairwise comparison results. Next is a real example of one of these tables (concretely, the result of running plaggie with a minimum match length of 6):



	InfixConverter/0_orig	InfixConverter/1_artifice	InfixConverter/test_0_orig_no_krakatau	InfixConverter/test_0_orig_no_procyon	InfixConverter/test_0_orig_pg_krakatau	InfixConverter/test_0_orig_pg_procyon	InfixConverter/test_1_artifice_no_krakatau	InfixConverter/test_1_artifice_no_procyon	InfixConverter/test_1_artifice_pg_krakatau	InfixConverter/test_1_artifice_pg_procyon	SqrtAlgorithm/0_orig	SqrtAlgorithm/1_artifice	SqrtAlgorithm/test_0_orig_no_krakatau	SqrtAlgorithm/test_0_orig_no_procyon	SqrtAlgorithm/test_0_orig_pg_krakatau	SqrtAlgorithm/test_0_orig_pg_procyon	SqrtAlgorithm/test_1_artifice_no_krakatau	SqrtAlgorithm/test_1_artifice_no_procyon	SqrtAlgorithm/test_1_artifice_pg_krakatau	SqrtAlgorithm/test_1_artifice_pg_procyon	hanoi/0_orig	hanoi/1_artifice	hanoi/test_0_orig_no_krakatau	hanoi/test_0_orig_no_procyon	hanoi/test_0_orig_pg_krakatau	hanoi/test_0_orig_pg_procyon	hanoi/test_1_artifice_no_krakatau	hanoi/test_1_artifice_no_procyon	hanoi/test_1_artifice_pg_krakatau	hanoi/test_1_artifice_pg_procyon	queens/0_orig	queens/1_artifice	queens/test_0_orig_no_krakatau	queens/test_0_orig_no_procyon	queens/test_0_orig_pg_krakatau	queens/test_0_orig_pg_procyon	queens/test_1_artifice_no_krakatau	queens/test_1_artifice_no_procyon	queens/test_1_artifice_pg_krakatau	queens/test_1_artifice_pg_procyon	square/0_orig	square/1_artifice	square/test_0_orig_no_krakatau	square/test_0_orig_no_procyon	square/test_0_orig_pg_krakatau	square/test_0_orig_pg_procyon	square/test_1_artifice_no_krakatau	square/test_1_artifice_no_procyon	square/test_1_artifice_pg_krakatau	square/test_1_artifice_pg_procyon
InfixConverter/0_orig	100	75	8	75	6	63	8	75	8	63	14	15	9	15	9	15	9	17	9	18	12	11	17	11	17	11	17	11	17	11	9	8	13	9	8	22	13	9	8	22	4	0	0	4	2	9	0	4	2	9
InfixConverter/1_artifice	81	100	10	60	8	44	10	60	8	44	14	18	7	15	7	15	7	17	7	18	12	18	17	11	17	11	17	11	17	11	9	14	13	9	8	16	13	9	8	16	0	7	6	0	5	7	6	0	5	7
InfixConverter/test_0_orig_no_krakatau	14	16	100	31	92	11	100	31	91	11	9	8	23	13	24	8	23	13	24	7	8	8	22	11	19	10	22	11	19	10	0	6	6	0	6	0	6	0	6	0	0	3	15	0	33	3	15	0	33	3
InfixConverter/test_0_orig_no_procyon	65	48	16	100	8	90	16	100	9	90	6	6	7	10	7	10	7	10	7	10	14	11	14	14	15	11	14	14	15	11	9	8	8	9	8	16	8	9	8	16	0	0	0	0	2	6	0	0	2	6
InfixConverter/test_0_orig_pg_krakatau	8	11	79	14	100	16	79	14	100	16	9	6	21	13	21	8	21	13	21	7	5	5	13	8	13	8	13	8	13	8	0	6	0	0	6	0	0	0	6	0	0	3	15	0	33	3	15	0	33	3
InfixConverter/test_0_orig_pg_procyon	43	28	4	71	8	100	4	71	8	100	6	6	4	10	5	10	4	10	5	10	9	8	5	9	5	9	5	9	5	9	9	8	13	9	8	16	13	9	8	16	0	0	0	0	0	3	0	0	0	3
InfixConverter/test_1_artifice_no_krakatau	14	16	100	31	92	11	100	31	91	11	9	8	23	13	24	8	23	13	24	7	8	8	22	11	19	10	22	11	19	10	0	6	6	0	6	0	6	0	6	0	0	3	15	0	33	3	15	0	33	3
InfixConverter/test_1_artifice_no_procyon	65	48	16	100	8	90	16	100	9	90	6	6	7	10	7	10	7	10	7	10	14	11	14	14	15	11	14	14	15	11	9	8	8	9	8	16	8	9	8	16	0	0	0	0	2	6	0	0	2	6
InfixConverter/test_1_artifice_pg_krakatau	12	11	78	14	100	16	78	14	100	16	9	6	21	13	21	8	21	13	21	7	5	5	13	8	13	8	13	8	13	8	0	6	0	0	6	0	0	0	6	0	0	3	18	0	33	3	18	0	33	3
InfixConverter/test_1_artifice_pg_procyon	43	28	4	71	8	100	4	71	8	100	6	6	4	10	5	10	4	10	5	10	9	8	5	9	5	9	5	9	5	9	9	8	13	9	8	16	13	9	8	16	0	0	0	0	0	3	0	0	0	3
SqrtAlgorithm/0_orig	17	16	7	9	8	12	7	9	8	12	100	54	25	67	21	69	25	63	21	64	3	3	9	6	9	6	9	6	9	6	7	0	6	6	7	14	6	6	7	14	23	8	4	20	10	18	4	20	10	16
SqrtAlgorithm/1_artifice	21	24	7	9	6	12	7	9	6	12	60	100	28	51	25	45	28	51	25	45	3	5	9	6	9	6	9	6	9	6	0	17	6	0	13	7	6	0	13	7	16	35	10	13	15	13	10	13	15	13
SqrtAlgorithm/test_0_orig_no_krakatau	16	11	25	14	26	12	25	14	26	12	35	35	100	34	89	30	100	36	89	31	3	3	10	9	10	8	10	9	10	8	0	0	12	0	19	13	12	0	19	13	4	3	23	4	35	10	23	4	35	10
SqrtAlgorithm/test_0_orig_no_procyon	17	16	9	14	11	17	9	14	11	17	62	43	22	100	19	98	22	94	19	92	5	5	9	11	9	11	9	11	9	11	0	0	6	6	7	7	6	6	7	7	27	5	4	26	12	28	4	26	12	29
SqrtAlgorithm/test_0_orig_pg_krakatau	16	11	24	14	26	12	24	14	26	12	29	32	87	29	100	29	87	31	100	31	0	0	8	6	8	6	8	6	8	6	0	0	6	0	19	13	6	0	19	13	4	3	21	4	34	11	21	4	34	11
SqrtAlgorithm/test_0_orig_pg_procyon	17	16	5	14	6	17	5	14	6	17	61	36	19	95	19	100	19	89	19	94	3	3	5	6	5	6	5	6	5	6	0	0	6	0	7	7	6	0	7	7	23	5	4	23	11	28	4	23	11	29
SqrtAlgorithm/test_1_artifice_no_krakatau	16	11	25	14	26	12	25	14	26	12	35	35	100	34	89	30	100	36	89	31	3	3	10	9	10	8	10	9	10	8	0	0	12	0	19	13	12	0	19	13	4	3	23	4	35	10	23	4	35	10
SqrtAlgorithm/test_1_artifice_no_procyon	21	19	9	14	11	17	9	14	11	17	61	44	24	99	22	97	24	100	22	98	5	5	9	11	9	11	9	11	9	11	0	0	6	6	7	7	6	6	7	7	19	3	4	19	12	21	4	19	12	22
SqrtAlgorithm/test_1_artifice_pg_krakatau	16	11	24	14	26	12	24	14	26	12	29	32	87	29	100	29	87	31	100	31	0	0	8	6	8	6	8	6	8	6	0	0	6	0	19	13	6	0	19	13	4	3	21	4	34	11	21	4	34	11
SqrtAlgorithm/test_1_artifice_pg_procyon	21	19	5	14	6	17	5	14	6	17	60	38	21	94	21	99	21	95	21	100	3	3	5	6	5	6	5	6	5	6	0	0	6	0	7	7	6	0	7	7	16	3	4	16	11	21	4	16	11	22
hanoi/0_orig	17	15	7	23	5	18	7	23	5	18	3	3	2	6	0	3	2	6	0	3	100	85	47	78	48	76	47	78	48	76	6	6	6	13	6	12	6	13	6	12	9	5	5	9	0	6	5	9	0	6
hanoi/1_artifice	17	24	7	19	5	18	7	19	5	18	3	5	2	6	0	3	2	6	0	3	88	100	43	69	44	67	43	69	44	67	6	6	6	6	6	6	6	6	6	6	6	11	5	6	0	3	5	6	0	3
hanoi/test_0_orig_no_krakatau	29	27	23	29	16	14	23	29	16	14	13	11	10	13	8	7	10	13	8	7	59	51	100	74	100	80	100	74	100	80	6	6	17	13	12	6	17	13	12	6	3	3	8	3	9	6	8	3	9	6
hanoi/test_0_orig_no_procyon	16	14	9	23	8	18	9	23	8	18	6	6	7	14	5	7	7	13	5	7	78	67	60	100	61	97	60	100	61	97	7	7	7	7	7	7	7	7	7	7	9	8	8	9	5	6	8	9	5	6
hanoi/test_0_orig_pg_krakatau	29	27	20	29	15	14	20	29	15	14	13	11	10	13	8	7	10	13	8	7	59	51	98	74	100	80	98	74	100	80	6	6	17	13	6	6	17	13	6	6	3	3	8	3	7	6	8	3	7	6
hanoi/test_0_orig_pg_procyon	16	14	9	19	8	18	9	19	8	18	6	6	7	13	5	7	7	13	5	7	76	65	64	97	66	100	64	97	66	100	7	7	7	7	7	7	7	7	7	7	9	8	8	9	4	6	8	9	4	6
hanoi/test_1_artifice_no_krakatau	29	27	23	29	16	14	23	29	16	14	13	11	10	13	8	7	10	13	8	7	59	51	100	74	100	80	100	74	100	80	6	6	17	13	12	6	17	13	12	6	3	3	8	3	9	6	8	3	9	6
hanoi/test_1_artifice_no_procyon	16	14	9	23	8	18	9	23	8	18	6	6	7	14	5	7	7	13	5	7	78	67	60	100	61	97	60	100	61	97	7	7	7	7	7	7	7	7	7	7	9	8	8	9	5	6	8	9	5	6
hanoi/test_1_artifice_pg_krakatau	29	27	20	29	15	14	20	29	15	14	13	11	10	13	8	7	10	13	8	7	59	51	98	74	100	80	98	74	100	80	6	6	17	13	6	6	17	13	6	6	3	3	8	3	7	6	8	3	7	6
hanoi/test_1_artifice_pg_procyon	16	14	9	19	8	18	9	19	8	18	6	6	7	13	5	7	7	13	5	7	76	65	64	97	66	100	64	97	66	100	7	7	7	7	7	7	7	7	7	7	9	8	8	9	4	6	8	9	4	6
queens/0_orig	6	5	0	6	0	8	0	6	0	8	3	0	0	0	0	0	0	0	0	0	3	3	2	3	2	3	2	3	2	3	100	60	36	76	28	38	36	76	28	38	11	0	4	11	3	0	4	11	3	3
queens/1_artifice	5	9	2	6	3	7	2	6	3	7	0	8	0	0	0	0	0	0	0	0	3	3	2	3	2	3	2	3	2	3	63	100	65	44	45	30	65	44	45	30	3	18	5	3	2	0	5	3	2	0
queens/test_0_orig_no_krakatau	9	8	2	6	0	13	2	6	0	13	3	3	4	3	2	3	4	3	2	3	3	3	6	3	6	3	6	3	6	3	37	64	100	49	66	37	100	49	66	37	4	4	12	4	11	3	12	4	11	3
queens/test_0_orig_no_procyon	6	5	0	6	0	8	0	6	0	8	3	0	0	3	0	0	0	3	0	0	5	3	4	3	4	3	4	3	4	3	72	40	45	100	33	62	45	100	33	62	26	3	4	26	3	7	4	26	3	10
queens/test_0_orig_pg_krakatau	5	5	2	6	3	7	2	6	3	7	3	6	7	4	7	4	7	4	7	4	3	3	4	3	2	3	4	3	2	3	28	44	66	35	100	39	66	35	100	39	4	5	14	4	18	0	14	4	18	0
queens/test_0_orig_pg_procyon	14	9	0	11	0	14	0	11	0	14	7	3	4	4	5	4	4	4	5	4	5	3	2	3	2	3	2	3	2	3	37	28	35	64	38	100	35	64	38	100	8	0	0	8	2	24	0	8	2	20
queens/test_1_artifice_no_krakatau	9	8	2	6	0	13	2	6	0	13	3	3	4	3	2	3	4	3	2	3	3	3	6	3	6	3	6	3	6	3	37	64	100	49	66	37	100	49	66	37	4	4	12	4	11	3	12	4	11	3
queens/test_1_artifice_no_procyon	6	5	0	6	0	8	0	6	0	8	3	0	0	3	0	0	0	3	0	0	5	3	4	3	4	3	4	3	4	3	72	40	45	100	33	62	45	100	33	62	26	3	4	26	3	7	4	26	3	10
queens/test_1_artifice_pg_krakatau	5	5	2	6	3	7	2	6	3	7	3	6	7	4	7	4	7	4	7	4	3	3	4	3	2	3	4	3	2	3	28	44	66	35	100	39	66	35	100	39	4	5	14	4	18	0	14	4	18	0
queens/test_1_artifice_pg_procyon	14	9	0	11	0	14	0	11	0	14	7	3	4	4	5	4	4	4	5	4	5	3	2	3	2	3	2	3	2	3	37	28	35	64	38	100	35	64	38	100	8	0	0	8	2	24	0	8	2	20
square/0_orig	4	0	0	0	0	0	0	0	0	0	22	14	3	28	3	25	3	19	3	16	8	5	2	8	2	8	2	8	2	8	21	6	8	52	8	16	8	52	8	16	100	44	20	92	6	67	20	92	6	66
square/1_artifice	0	9	2	0	3	0	2	0	3	0	9	34	2	6	2	7	2	3	2	3	5	10	2	8	2	8	2	8	2	8	0	37	9	6	12	0	9	6	12	0	50	100	48	44	20	30	48	44	20	30
square/test_0_orig_no_krakatau	0	7	12	0	14	0	12	0	14	0	4	10	18	4	17	4	18	4	17	4	5	5	7	8	6	8	7	8	6	8	8	11	26	8	31	0	26	8	31	0	23	49	100	21	62	10	100	21	62	10
square/test_0_orig_no_procyon	4	0	0	0	0	0	0	0	0	0	19	11	3	27	3	24	3	19	3	16	8	5	2	8	2	8	2	8	2	8	21	6	8	52	8	16	8	52	8	16	92	38	18	100	6	76	18	100	6	76
square/test_0_orig_pg_krakatau	4	7	35	4	40	0	35	4	41	0	14	19	34	19	34	17	34	18	34	17	0	0	10	6	8	5	10	6	8	5	8	6	31	8	49	6	31	8	49	6	9	26	79	9	100	12	79	9	100	12
square/test_0_orig_pg_procyon	12	8	2	9	3	6	2	9	3	6	19	12	8	31	8	32	8	23	8	23	5	3	4	5	4	5	4	5	4	5	0	0	6	16	0	51	6	16	0	51	72	29	10	83	9	100	10	83	9	100
square/test_1_artifice_no_krakatau	0	7	12	0	14	0	12	0	14	0	4	10	18	4	17	4	18	4	17	4	5	5	7	8	6	8	7	8	6	8	8	11	26	8	31	0	26	8	31	0	23	49	100	21	62	10	100	21	62	10
square/test_1_artifice_no_procyon	4	0	0	0	0	0	0	0	0	0	19	11	3	27	3	24	3	19	3	16	8	5	2	8	2	8	2	8	2	8	21	6	8	52	8	16	8	52	8	16	92	38	18	100	6	76	18	100	6	76
square/test_1_artifice_pg_krakatau	4	7	35	4	40	0	35	4	41	0	14	19	34	19	34	17	34	18	34	17	0	0	10	6	8	5	10	6	8	5	8	6	31	8	49	6	31	8	49	6	9	26	79	9	100	12	79	9	100	12
square/test_1_artifice_pg_procyon	12	8	2	9	3	6	2	9	3	6	17	12	8	32	8	33	8	24	8	24	5	3	4	5	4	5	4	5	4	5	7	0	6	23	0	43	6	23	0	43	71	29	10	83	9	100	10	83	9	100

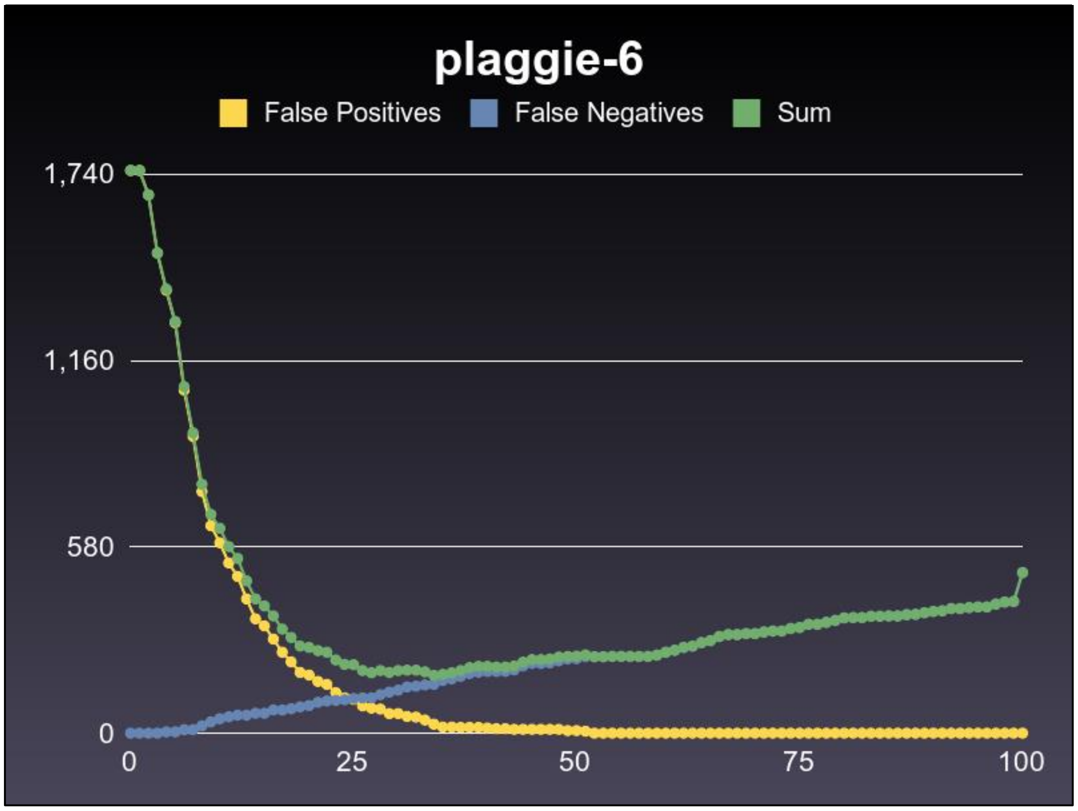
It is quite obvious from just quick looking at the table, that this plagiarism detector with this particular parameters is somehow able to identify plagiarism. All the test programs that belong to the same group (that is, all the obfuscated versions of the same program) produce higher numbers between them than between them and programs in other groups. Since programs in the same group appear consecutively in the table rows and columns, square shaped patterns can be appreciated in the table, produced by higher values with more digits.

The problem is that, as explained earlier, we need to define a number that cuts out what we will consider as a positive and what is a negative, that is, the threshold value. To find it, we’ve developed a script that identifies the program groups in the CSV, then iterates all possible threshold values from 0 to 100 classifying all pairwise comparisons into positives or negatives, and those into errors or mistakes, finally being able to build a confusion matrix for each threshold. This is the confusion matrix produced for the table above:



	A	B	C	D	E	F	G	H	I	J	K	L
1	Threshold	False Positives	False Negatives	Sum	Threshold	False Positives	alse Negative	Sum	Threshold	False Positives	False Negatives	Sum
2	0	1752	0	1752	33	41	150	191	66	0	301	301
3	1	1752	0	1752	34	28	152	180	67	0	307	307
4	2	1676	0	1676	35	19	164	183	68	0	307	307
5	3	1495	0	1495	36	19	169	188	69	0	310	310
6	4	1379	4	1383	37	18	177	195	70	0	310	310
7	5	1278	4	1282	38	18	186	204	71	0	315	315
8	6	1068	11	1079	39	18	190	208	72	0	318	318
9	7	924	11	935	40	16	192	208	73	0	318	318
10	8	752	23	775	41	14	192	206	74	0	326	326
11	9	646	35	681	42	14	192	206	75	0	329	329
12	10	593	45	638	43	12	197	209	76	0	339	339
13	11	530	51	581	44	12	209	221	77	0	339	339
14	12	488	56	544	45	12	217	229	78	0	345	345
15	13	418	56	474	46	12	217	229	79	0	351	351
16	14	356	62	418	47	12	219	231	80	0	359	359
17	15	334	62	396	48	12	225	237	81	0	360	360
18	16	293	72	365	49	8	231	239	82	0	360	360
19	17	252	72	324	50	8	232	240	83	0	364	364
20	18	222	76	298	51	6	238	244	84	0	364	364
21	19	189	82	271	52	0	238	238	85	0	365	365
22	20	181	86	267	53	0	238	238	86	0	365	365
23	21	161	96	257	54	0	239	239	87	0	369	369
24	22	152	100	252	55	0	239	239	88	0	370	370
25	23	126	102	228	56	0	239	239	89	0	375	375
26	24	110	104	214	57	0	239	239	90	0	379	379
27	25	105	108	213	58	0	239	239	91	0	381	381
28	26	85	110	195	59	0	243	243	92	0	388	388
29	27	78	110	188	60	0	252	252	93	0	388	388
30	28	75	120	195	61	0	258	258	94	0	391	391
31	29	61	128	189	62	0	267	267	95	0	393	393
32	30	61	134	195	63	0	271	271	96	0	393	393
33	31	52	144	196	64	0	282	282	97	0	402	402
34	32	50	146	196	65	0	288	288	98	0	408	408
35									99	0	410	410
36									100	0	500	500

Additionally, all confusion matrixes are graphed for easy visual inspection. This is the graph produced from the confusion matrix above:



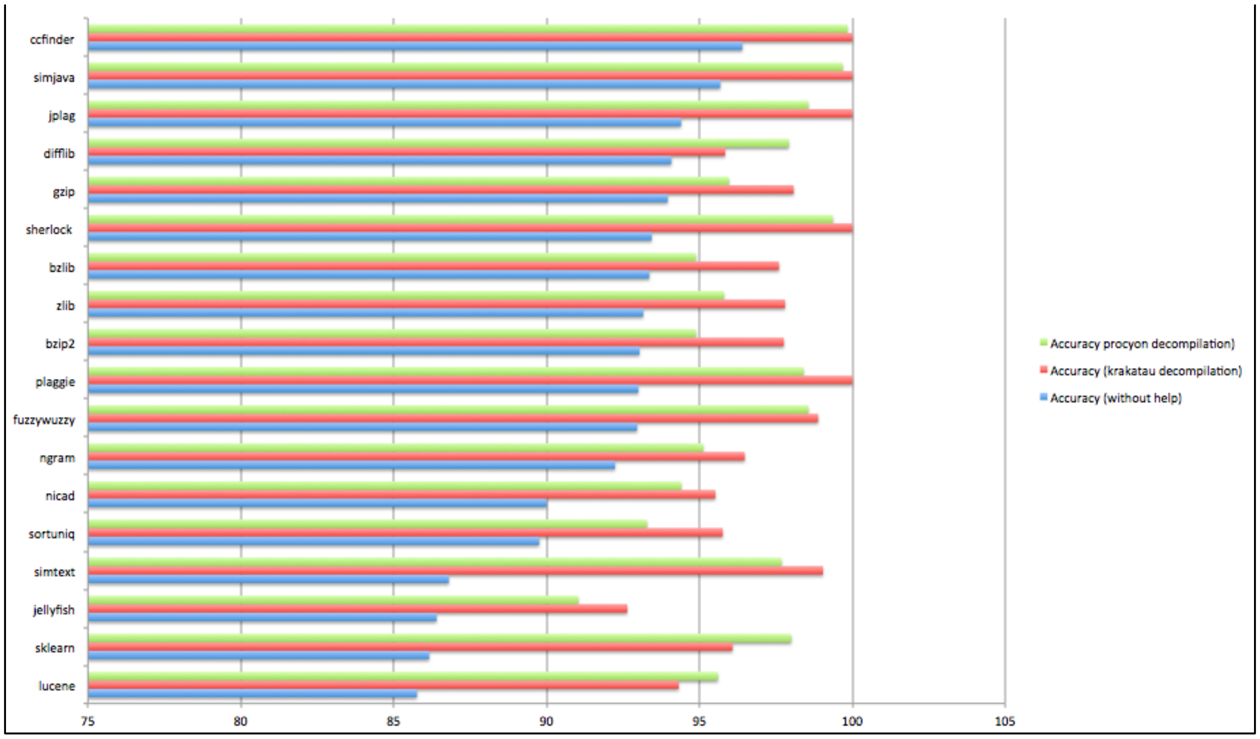
Finally, the threshold that produces less mistakes is logged with the mistakes it produced. In this case, the log reads:

“plaggie-6: Best threshold is 34 with a total number of 180 mistakes”

Using these logs we can build a graph that compares the best scenario performance for all the tools.

4.3 Results

After applying the process described at 4.2 for all the plagiarism detection tools on all the test programs we finally obtain the results we were looking for:



The first noticeable thing is that the attempt to help the plagiarism detection tools described at 3.4 has been a success, especially on the tools that weren't preprocessing the files, but also the ones that do. Accuracy has improved a 5.35% when preprocessing the files via compilation and decompilation with Procyon, and a 6.37% when using Krakatau. What's more, when using Krakatau as a decompiler for the preprocessing, some tools (Ccfinder, Simjava, Jplag, Sherlock and Plaggie) performed perfectly and didn't make a single mistake when classifying the 2500 pairs as plagiarism or non-plagiarism.

4.4 Why preprocessing made the difference

Most of the tools that scored high in accuracy without help implement some kind of preprocessing of their own. However, no program understands the complexity and the structure of a source code file as a compiler/decompiler does. All of the mistakes made by tools that later performed flawlessly were false negatives produced by obfuscations actually managing to fool them. However, by processing source code files in this way, the number of false negatives got reduced to zero. Next is a example on how the preprocessing helped normalize the code a lot, the constructor of the MagicSquare class. On the top we have the comparison the tools had to do without help, and below the one after preprocessing with Krakatau. On the left is the original file and on the right the one compiled, obfuscated with artifice and ProGuard, and decompiled with Procyon.

```
public MagicSquare ( int n ) {
    square = new int[n][n];
    for ( int i = 0; i < n; i++ )
      for ( int j = 0; j < n; j++ ) {
        square[i][j] = 0;
      }
    totalSqs = n * n;
    possible = new boolean[totalSqs];
    for ( int i = 0; i < totalSqs; i++ ) {
      possible[i] = true;
    }
    sum = n * ( n * n + 1 ) / 2;
    numsquares = 0;
  }
```

```
private MagicSquare ( int i ) {
    super();
    this.a = new int[3][3];
    for ( i = 0; i < 3; ++i ) {
      for ( int j = 0; j < 3; ++j ) {
        this.a[i][j] = 0;
      }
    }
    this.c = 9;
    this.b = new boolean[this.c];
    for ( int k = 0; k < this.c; ++k ) {
      this.b[k] = true;
    }
    this.d = 15;
    this.e = 0;
  }
```

```
  public MagicSquare ( int i ) {
    super();
    this.square = new int[i][i];
    int i0 = 0;
    while ( i0 < i ) {
      int i1 = 0;
      while ( i1 < i ) {
        this.square[i0][i1] = 0;
        i1 = i1 + 1;
      }
      i0 = i0 + 1;
    }
    this.totalSqs = i * i;
    this.possible = new
boolean[this.totalSqs];
    int i2 = 0;
    while ( i2 < this.totalSqs ) {
      this.possible[i2] = true;
      i2 = i2 + 1;
    }
    this.sum = i * ( i * i + 1 ) / 2;
    this.numsquares = 0;
  }
```

```
  private MagicSquare ( int i ) {
    super();
    this.a = new int[3][3];
    int i0 = 0;
    while ( i0 < 3 ) {
      int i1 = 0;
      while ( i1 < 3 ) {
        this.a[i0][i1] = 0;
        i1 = i1 + 1;
      }
      i0 = i0 + 1;
    }
    this.c = 9;
    this.b = new boolean[this.c];
    int i2 = 0;
    while ( i2 < this.c ) {
      this.b[i2] = true;
      i2 = i2 + 1;
    }
    this.d = 15;
    this.e = 0;
  }
```

It is quite obvious that aside from the fact that the obfuscators got rid of the variable names and applied some inlining with constant values, the structure of the code is almost identical when preprocessing with this method.

5. Conclusion

While source code obfuscation isn't a magic solution to bypass plagiarism detectors, they definitely impact their performance. It is worth nothing that the obfuscations used to perform this project were entirely automated and therefore probably of a lower grade than a real plagiarist would or could accomplish. Moreover a perfect tuning of the parameters was assumed, which is surely not the case in reality.

Having a 95% accuracy in a best possible case scenario is not acceptable in a lot of contexts, and thus there is a real need for the development and research of new tools to detect plagiarism regardless of source code obfuscation.

The results of this project suggest that source obfuscation robustness has a very strong correlation with the preprocessing that is applied to the source code before comparing. The tools that have some kind of preprocessing clearly performed better than those that don't, and upon applying a very strong preprocessing technique all the tools experimented a very significant boost in their performance, with some of them not making a single mistake.

The final conclusion then, is that while current plagiarism detection algorithms are suitable for today's environment, the preprocessing techniques are not. In this project a compilation/decompilation process is suggested and implemented as a preprocessing technique with good results, and therefore the authors hope to see this implemented in a plagiarism detection tool in the future.

6. Bibliography

Cosma, G. and Joy, M. (2008) 'Towards a definition of source-code plagiarism' Education, IEEE Transactions on, 51 (2), pp. 195--200.

Bull, J., Collins, C., Coughlin, E., Sharp, D. and Square, P. (2001) 'Technical review of plagiarism detection software report' University of Luton. JISC publication.

Fiducia, N. and Briskman, S. (2013) 'When Two Worlds Collide: The Oracle And Google Dispute' Mondaq.com. [online] Available at: http://www.mondaq.com/unitedstates/x/271942/ [Accessed: 13 Feb 2014].

Roy, C. K., Cordy, J. R. and Koschke, R. (2009) 'Comparison and evaluation of code clone detection techniques and tools: A qualitative approach' Science of Computer Programming, 74 (7), pp. 470--495.

Schulze, S. and Meyer, D. (2013) 'On the robustness of clone detection to code obfuscation' IWSC 2013, pp. 62 - 68.

Meyer, D. and Schulze, S. (2012) 'Analyzing the Robustness of Clone Detection Tools Regarding Code Obfuscation'.

Lafortune, E. (2013) 'ProGuard' Proguard.sourceforge.net. [online] Available at: http://proguard.sourceforge.net/ [Accessed: 13 Feb 2014].

AOSP. (2013) 'ProGuard | Android Developers' developer.android.com. [online] Available at: https://developer.android.com/tools/help/proguard.html [Accessed: 13 Feb 2014].

Grosse, R. (2013) 'Krakatau Bytecode Tools' Krakatau Bytecode Tools. [online] Available at: https://raw.github.com/Storyyeller/Krakatau/master/README.TXT [Accessed: 13 Feb 2014].

Wise, M. J. (1993) 'String similarity via greedy string tiling and running Karp-Rabin matching' Online Preprint, Dec, 119.

Prechelt, L., Malpohl, G. and Philippsen, M. (2002) 'Finding plagiarisms among a set of programs with JPlag' J. UCS, 8 (11), p. 1016.

Kamiya, T., Kusumoto, S. and Inoue, K. (2002) 'CCFinder: a multilinguistic token-based code clone detection system for large scale source code' Software Engineering, IEEE Transactions on, 28 (7), pp. 654--670.

Cordy, J. R. and Roy, C. K. (2011) 'The NiCad clone detector' pp. 219--220.

Hage, J., Rademaker, P. and Vugt, N. (2010) 'A comparison of plagiarism detection tools' Utrecht University. Utrecht, The Netherlands, p. 28.

Warwick. (2009) 'BOSS Online Submission System' dcs.warwick.ac.uk. [online] Available at: http://www.dcs.warwick.ac.uk/boss/history.php [Accessed: 13 Feb 2014].

Grune, D. and Vakgroep, M. (1989) 'Detecting copied submissions in computer science workshops' Informatica Faculteit Wiskunde \& Informatica, Vrije Universiteit, 9.

Cebrian, M., Alfonseca, M. and Ortega, A. (2009) 'Towards the validation of plagiarism detection tools by means of grammar evolution' Evolutionary Computation, IEEE Transactions on, 13 (3), pp. 477--485.

Salton, G., Wong, A. and Yang, C. (1975) 'A vector space model for automatic indexing' Communications of the ACM, 18 (11), pp. 613--620.

Si, A., Leong, H. V. and Lau, R. W. (1997) 'CHECK: a document plagiarism detection system' pp. 70--77.

W. Ratcliff, J. and E. Metzener, D. (1982) 'Pattern Matching: The Gestalt Approach' Dr. Dobbs Journal, pp. 46--51.

Kontogiannis, K. A., Demori, R., Merlo, E., Galler, M. and Bernstein, M. (1996) 'Pattern matching for clone and concept detection' Springer, pp. 77--108.

Cohen, W. W., Ravikumar, P. and Fienberg, S. E. (2003) ''A Comparison of String Distance Metrics for Name-Matching Tasks.'', pp. 73--78.

Rahman,, S. and Puspitodjati, S. (2009) 'SIMILARITY COMPARISON APPLICATIONS BETWEEN DOCUMENT USING JARO Winkler DISTANCE algorithm TO DETECT SCIENTIFIC WRITING PLAGIARISM IN GUNADARMA UNIVERSITY'.

Kondrak, G. (2005) 'N-gram similarity and distance' SPIRE'05, pp. 115--126.

Barrón-Cedeño, A. and Rosso, P. (2009) 'On automatic plagiarism detection based on n-grams comparison' ECIR '09, pp. 696--700.

Hamilton, H. (2012) 'Confusion Matrix' www2.cs.uregina.ca. [online] Available at: http://www2.cs.uregina.ca/~dbd/cs831/notes/confusion_matrix/confusion_matrix.html [Accessed: 13 Feb 2014].