UCL DEPARTMENT OF
COMPUTER SCIENCE

Research Note
RN/17/04

A Comparison of Code Similarity Analysers

20 February 2017

Chaiyong Ragkhitwetsagul Jens Krinke David Clark

Abstract

Source code analysis to detect code cloning, code plagiarism, and code reuse suffers from the problem of pervasive code modifications, i.e. transformations that may have a global effect. We compare 30 similarity detection techniques and tools against pervasive code modifications. We evaluate the tools using four experimental scenarios for Java source code. These are (1) pervasive modifications created with tools for source code and bytecode obfuscation, (2) source code normalisation through compilation and decompilation using different decompilers, (3) reuse of optimal configurations over different data sets, and (4) tool evaluation using ranked-based measures. Our experimental results show that highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures. Our study strongly validates the use of compilation/decompilation as a normalisation technique. Its use reduced false classifications to zero for three of the tools. We also demonstrate that optimal configurations are very sensitive to a specific data set. Our findings show that after directly applying optimal configurations derived from one data set to another, the tools perform poorly on the new data set. This broad, thorough study is the largest in existence and potentially an invaluable guide for future users of similarity detection in source code.

Keywords Empirical study · Code similarity measurement · Clone detection · Plagiarism detection · Parameter optimisation

1 Introduction

Assessing source code similarity is a fundamental activity in software engineering and it has many applications. These include clone detection, the problem of locating duplicated code fragments; plagiarism detection; software copyright infringement; and code search, in which developers search for similar implementations. While that list covers the more common applications, similarity assessment is used in many other areas, too. Examples include finding similar bug fixes (Hartmann et al, 2010), identifying cross-cutting concerns (Bruntink et al, 2005), program comprehension (Maletic and Marcus, 2001), code recommendation (Holmes and Murphy, 2005), and example extraction (Moreno et al, 2015).

The assessment of source code similarity has a co-evolutionary relationship with the modifications made to the code at the point of its creation. In this paper we consider not only local transformations but also pervasive modifications, such as changes in layout or renaming of identifiers, changes that affect the code globally. Loosely, these are code transformations that arise in the course of code cloning, software plagiarism, and software evolution, but exclude strong obfuscation (Collberg et al, 1997). In code reuse by code cloning, which occurs through copying and pasting a fragment from one place to another, the copied code is often modified to suit the new environment (Roy et al, 2009). Modifications include formatting changes and identifier renaming (Type I and II clones), structural changes, e.g. `if` to `case` or `while` to `for`, or insertions or deletions (Type III clones) (Davey et al, 1995). Likewise, software plagiarisers copy source code of a program and modify it to avoid being caught (Daniela et al, 2012). Moreover, source code is modified during software evolution (Pate et al, 2013). Therefore, most clone or plagiarism detection tools and techniques tolerate different degrees of change and still identify cloned or plagiarised fragments. However, while they usually have no problem in the presence of local or confined modifications, pervasive modifications that transform whole files or systems remain a challenge (Roy and Cordy, 2009).

This work is motivated by the question: *"When source code is pervasively modified, which similarity detection techniques or tools get the most accurate results?"* To answer this question, we provide a thorough evaluation of the performance of the current state-of-the-art similarity detection techniques on pervasively modified code. The study presented in this paper is the largest extant study on source code similarity and covers the widest range of techniques and tools. Previous studies, e.g. on the accuracy of clone detection tools (Bellon et al, 2007; Roy et al, 2009; Svajlenko and Roy, 2014) and of plagiarism detection tools (Hage et al, 2010), were mainly focused on a single technique or tool, or on a single domain.

Our aim is to provide a foundation for the appropriate choice of a similarity detection technique or tool for a given application based on a thorough evaluation of strengths and weaknesses. Choosing the wrong technique or tool with which to measure software similarity or even just choosing the wrong parameters may have detrimental consequences.

We have selected as many techniques for source code similarity measurement as possible, 30 in all, covering techniques specifically designed for clone and plagiarism detection, plus the normalised compression distance, string matching, and information retrieval. In general, the selected tools require the optimisation of their parameters as these can affect the tools' execution behaviours and consequently their results. A previous study regarding parameter optimisation (Wang et al, 2013) has explored only a small set of parameters of clone detectors. Therefore, while including more tools in this study, we have also searched a wider range of configurations for each tool, studied their impact, and discovered the best configurations for each data set in our experiments.

Clone and plagiarism detection use intermediate representations like token streams or abstract syntax trees or other transformations like pretty printing or comment removal to achieve a normalised representation (Roy et al, 2009). We integrated compilation and decompilation as a normalisation pre-process step for similarity detection and evaluated its effectiveness.

This paper makes the following primary contributions:

1. A broad, thorough study of the performance of similarity tools and techniques: We compare a large range of 30 similarity detection techniques and tools using two experimental scenarios for Java source code in order to measure the techniques' performances and observe their behaviours. We apply several error measures including pair-based and query-based measures. The results show that highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures.

The results of the evaluation can be used by researchers as guidelines for selecting techniques and tools appropriate for their problem domain. Our study confirms both that tool configurations have strong effects on tool performance and that they are sensitive to particular data sets. Poorly chosen techniques or configurations can severely affect results.

2. Normalisation by decompilation: Our study confirms that compilation and decompilation as a pre-processing step can normalise pervasively modified source code and can greatly improve the effectiveness of similarity measurement techniques. Six of the similarity detection techniques and tools reported no false classifications once such normalisation was applied.

Compared to our previous work (Ragkhitwetsagul et al, 2016), we have expanded the study further by including three additional activities. First, we doubled the size of the generated data set from 5 original Java classes to 10 classes and re-evaluated the tools. This change made the number of pairwise comparisons quadratically increase from 2,500 to 10,000. With this expanded data set, we could better observe the tools' performances on pervasively modified source code. We found some differences in the tool rankings using the new data set when compared to the previous one. Second, besides source code with pervasive modifications, we compared the similarity analysers on an available data set containing reuse of boiler-plate code. Since boiler-plate code is inherently different from pervasively modified code and normally found in software development (Kapser and Godfrey, 2006), the findings give a guideline to choosing the right tools/techniques when measuring code similarity in the presence of boiler-plate code. Third, we investigated the effects of reusing optimal configurations from one data set on another data set. Our empirical results show that the optimal configurations are very sensitive to a specific data set and not suitable for reuse.

2 Background

2.1 Code Similarity Measurement

Since the 1970s, myriads of tools have been introduced to measure similarity of source code. They are used to tackle problems such as code clone detection, software licensing violation, and software plagiarism. The tools utilise different approaches to compute the similarity of two programs. We can classify them into metrics-based, text-based, token-based, tree-based, and graph-based approaches. Early approaches to detect software similarity (Ottenstein, 1976; Donaldson et al, 1981; Grier, 1981; Berghel and Sallach, 1984; Faidhi and Robinson, 1987) are based on metrics or software measures. One of the early code similarity detection tools was created by Ottenstein (1976) and was based on Halstead complexity measures (Halstead, 1977) and was able to discover a plagiarised pair out of 47 programs of students registered in a programming class. Unfortunately, the metrics-based approaches have been empirically found to be less effective in comparison with other, newer approaches (Kapser and Godfrey, 2003).

Text-based approaches perform similarity checking based on comparing two string sequences of source code. They are able to locate exact copies of source code, while usually susceptible to finding similar code with syntactic and semantic modifications. Some supporting techniques are incorporated to handle syntactic changes such as variable renaming (Roy and Cordy, 2008). There are several code similarity analysers that compute textual similarity. One of the widely-used string similarity is to find a longest common subsequence (LCS) which is adopted by the NiCad clone detector (Roy and Cordy, 2008), Plague (Whale, 1990), the first version of YAP (Wise, 1992), and CoP (Luo et al, 2014). Other text-based tools with string matching algorithms other than LCS include, but not limited to, Duploc (Ducasse et al, 1999), Simian (Harris, 2015), and PMD's Copy/Paste Detector (CPD).

To take one step of abstraction up from literal code text, we can transform source code into tokens (i.e. words). A stream of tokens can be used as an abstract representation of a program. The level of abstraction level can be adjusted by the defining the types of tokens. Depending on how the tokens are defined, the token stream may normalise textual differences and captures only an abstracted sequence of a program. For example, if every word in a program is replaced by a `W` token, a statement `int x = 0;` will be similar to `String s = "Hi";` because they both share a token stream of `W W = W;`. Different similarity measurements such as suffix trees, string alignment, Jaccard similarity, etc., can be applied to sequences or sets of tokens. Tools that rely on tokens include Sherlock (Joy and Luck, 1999), BOSS (Joy

et al, 2005), Sim (Gitchell and Tran, 1999), YAP3 (Wise, 1996), JPlag (Prechelt et al, 2002), CCFinder (Kamiya et al, 2002), CP-Miner (Li et al, 2006), MOSS (Schleimer et al, 2003), Burrows et al. (Burrows et al, 2007), and the Source Code Similarity Detector System (SCSDS) (Duric and Gasevic, 2012). The token-based representation is widely used in source code similarity measurement and very efficient over a scale of millions SLOC. An example is the large-scale token-based clone detection tool SourcererCC (Sajnani et al, 2016).

Tree-based code similarity measurement can avoid issues of formatting and lexical differences and focus only on locating structural sameness between two programs. Abstract Syntax Trees (ASTs) are the widely-used structure when computing program similarity by finding similar subtrees between two ASTs. Its capability of comparing programs' structures allow tree-based tools to locate similar code with wider range of modifications such as added or deleted statements. However, tree-based similarity measures have a high computational complexity. The comparison of two ASTs with N nodes can have an upper bound of $O(N^3)$ (Baxter et al, 1998). Usually an optimising mechanism or approximation is included in the similarity computation to lower the computation time (Jiang et al, 2007b). A few examples of well-known tree-based tools include CloneDR (Baxter et al, 1998), and Deckard (Jiang et al, 2007b).

Graph-based structures are chosen when one wants to capture not only the structure but also the semantics of a program. However, like trees, graph similarity suffers from a high computational complexity. Algorithms for graph comparison are mostly NP-complete (Liu et al, 2006; Crussell et al, 2012; Krinke, 2001; Chae et al, 2013). In clone and plagiarism detection, a few specific types of graphs are used, e.g. program dependence graphs (PDG), or control flow graphs (CFG). Examples of code similarity analysers using graph-based approaches are the ones invented by Krinke (Krinke, 2001), Komondoor et al. (Komondoor and Horwitz, 2001), Chae et al. (Chae et al, 2013) and Chen et al (Chen et al, 2014). Although the tools demonstrated high precision and recall (Krinke, 2001), they suffer scalability issues (Bellon et al, 2007).

Code similarity measurement can not only be measured on source code but also on compiled code. Measuring similarity of compiled code is useful when the source code is absent or prohibited from access. Moreover, it can also capture dynamic behaviours of the programs by executing the compiled code. In the last few years, there have been several studies to discover cloned and plagiarised programs (especially mobile applications) based on compiled code (Chae et al, 2013; Chen et al, 2014; Gibler et al, 2013; Crussell et al, 2013; Tian et al, 2014; Tamada et al, 2004; Myles and Collberg, 2004; Lim et al, 2009; Zhang et al, 2012; McMillan et al, 2012; Luo et al, 2014; Zhang et al, 2014; Crussell et al, 2012).

Besides the text, token, tree, and graph-based approaches, there are several other alternative techniques adopted from other fields of research to code similarity measurement such as information theory, information retrieval, or data mining. These techniques show positive results and open more possibilities to this research area. Examples of these techniques include Software Bertillonage (Davies et al, 2013), Kolmogorov complexity (Li and Vitâanyi, 2008), Latent Semantic Indexing (LSI) (McMillan et al, 2012), and Latent Semantic Analysis (LSA) (Cosma and Joy, 2012).

2.2 Obfuscation and Deobfuscation

Obfuscation is a mechanism of making changes to a program while preserving its original functions. It originally aims to protect intellectual property of computer programs from reverse engineering or from malicious attack (Collberg et al, 2002) and can be achieved in both source and binary level. Many automatic code obfuscation tools are available nowadays both for commercial (e.g. Semantic Designs Inc.'s C obfuscator (Semantic Designs, 2016), Stunnix's obfuscators (Stunnix, 2016), Diablo (Maebe and Sutter, 2006)) and research purposes (Chow et al, 2001; Schulze and Meyer, 2013; Madou et al, 2006; Necula et al, 2002).

Given a program P, and the transformed program P', the definition of obfuscation transformations T is $P \xrightarrow{T} P'$ requiring P and P' to hold the same observational behaviour (Collberg et al, 1997). Specifically, *legal* obfuscation transformation requires: 1) if P fails to terminate or terminates with errors then P' may or may not terminate, and 2) P' must terminate if P terminates.

Generally, there are three approaches for obfuscation transformations: lexical (layout), control, and data transformation (Collberg et al, 2002, 1997). Lexical transformations can be achieved by renaming identifiers and formatting changes, while control transformations use more sophisticated methods such as embedding spurious branches and opaque

predicates which can be deducted only at runtime. Data transformations make changes to data structures and hence the source code difficult to reverse engineer. Similarly, binary-code obfuscators transform content of executable files.

Many obfuscation techniques have been invented and put to use in commercial obfuscators. Collberg and Myles (Collberg et al, 2003) introduce several reordering techniques (e.g. method parameters, basic block instructions, variables, and constants), splitting of classes, basic blocks, arrays, and also merging of method parameters, classes. These techniques are implemented in their tool, SandMark. Wang et al. (Wang et al, 2001) propose a sophisticated deep obfuscation method called *control flow flattening* which flattens the control flow graph by dividing normal contiguous statements into separated branches inside the same `switch` statement and controls the program behaviours by a dispatcher variable. It has been later used in a commercial tool called Cloakware. ProGuard (GuardSquare, 2015) is a Java bytecode obfuscator which performs obfuscation by removing existing names (e.g. class, method names), replace them with meaningless characters, and also gets rid of all debugging information from Java bytecode. Loco (Madou et al, 2006) is a binary obfuscator capable of performing obfuscation using control flow flattening and opaque predicates on selected fragments of code.

Deobfuscation is a method aiming at reversing the effects of obfuscation which can be achieved at either static and dynamic level. It can be useful in many aspects such as detection of obfuscated malwares (Nachenberg, 1996) or as a resiliency testing for a newly developed obfuscation method (Madou et al, 2006). While *surface obfuscation* such as variable renaming can be handled quite straightforward, *deep obfuscation* that makes large changes to the structure of the program (e.g. opaque predicates or control flow flattening) is much more difficult to reverse. However, it is not totally impossible. It has been shown that one can counter control flow flattening by either cloning the portions of added spurious code to separate them from the original execution path, or use static path feasibility analysis (Udupa et al, 2005).

2.3 Program Decompilation

Decompilation of a program generates high-level code from low-level code. It has several benefits including recovery of lost source code from compiled artefacts such as binary or bytecode, reverse engineering, finding similar applications (Chen et al, 2014). On the other hand, decompilation can also be used to create program clones by decompiling a program, making changes, and repacking it into a new program. An example of this malicious use of decompilation can be seen from a study by Chen et al. (Chen et al, 2014). They found that 13.51% of all applications from five different Android markets are clones. Gibler et al. (Gibler et al, 2013) discovered that these decompiled and cloned apps can divert ad impressions from the original app owners by 14% and divert potential users by 10%.

Many decompilers have been invented in the literature for various programming languages (Cifuentes and Gough, 1995; Proebsting and Watterson, 1997; Desnos and Gueguen, 2011; Mycroft, 1999; Breuer and Bowen, 1994). Several techniques are involved to successfully decompile a program. The decompiled source code may be different according to each particular decompiler. Conceptually, decompilers extract semantics of programs from their executables, then, with some heuristics, generate the source code based on this extraction. For example Krakatoa (Proebsting and Watterson, 1997), a Java decompiler, extracts expressions and type information from Java bytecode using symbolic execution, and creates a control flow graph (CFG) of the program representing the behaviour of the executable. Then, to generate source code, a sequencer arranges the nodes and creates an abstract syntax tree (AST) of the program. The AST is then simplified by rewriting rules and, finally, the resulting Java source code is created by traversing the AST.

It has been found that program decompilation has an additional benefit of code normalisation. An empirical study (Ragkhitwetsagul and Krinke, 2017) shows that, compared to clones in the original versions, additional clones were found after compilation/decompilation in three real-world software projects. Many of the newly discovered clone pairs contained several modifications which were causing difficulty for clone detectors. Compilation and decompilation canonicalise these changes and the clone pairs became very similar after decompilation.

3 Empirical Study

Our empirical study consists of four experimental scenarios covering different aspects and characteristics of source code similarity. Two experimental scenarios examined tool/technique performance on two different data sets to discover any strengths and weaknesses. These two are (1) experiments on the products of the two obfuscation tools and (2) experiments on an available data set for identification of reuse boiler-plate code (Flores et al, 2014). The third scenario examined the effectiveness of compilation/decompilation as a preprocessing normalisation strategy and the fourth evaluated the use of error measures from information retrieval for comparing tool performance without relying on a threshold value.

The empirical study aimed to answer the following research questions:

RQ1 (Performance comparison): *How well do current similarity detection techniques perform in the presence of pervasive source code modifications?* We compare 30 code similarity analysers using a data set of 1,000 pervasively modified source code.

RQ2 (Optimal configurations): *What are the best parameter settings and similarity thresholds for the techniques?* We exhaustively search wide ranges of the tools' parameter values to locate the ones that give optimal performances.

RQ3 (Normalisation by decompilation): *How much does compilation followed by decompilation as a pre-processing normalisation method improve detection results?* We apply compilation and decompilation to the data set before running the tools. We compare the performances before and after applying this normalisation.

RQ4 (Reuse of configurations): *Can we reuse optimal configurations from one data set in another data set effectively?* We apply the optimal configurations obtained from one data set to another data set and see if they still offer the tools' best performances.

RQ5 (Ranked Results): *How accurate are the results when only the top n results are retrieved?* Besides the set-based error measures normally used in clone and plagiarism detection evaluation, we also compare and report the tools' performances using rank-based measures adopted from information retrieval. This comparison has a practical benefit in terms of plagiarism detection.

3.1 Experimental Framework

The general framework of our study, as shown in Figure 1, consists of 5 main steps. In Step 1, we collect test data consisting of Java source code files. Next, the source files are transformed by applying pervasive modifications at source and bytecode level. In the third step, all original and transformed source files are normalised. A simple form of normalisation is pretty printing the source files which is used in similarity or clone detection (Roy and Cordy, 2008). We also use decompilation. In Step 4, the similarity detection tools are executed pairwise against the set of all normalised files, producing similarity reports for every pair. In the last step, the similarity reports are analysed.

In the analysis step, we extract a similarity value $\mathrm{sim}(x, y)$ from the report for every pair of files x, y, and based on the reported similarity, the pair is classified as being similar (reused code) or not according to some chosen threshold T. The set of similar pairs of files $\mathrm{Sim}(F)$ out of all files F is

$$\mathrm{Sim}(F) = \{(x, y) \in F \times F : \mathrm{sim}(x, y) > T\} \tag{1}$$

We selected data sets for which we know the ground truth, allowing decisions on whether a code pair is correctly classified as a similar pair (true positive, *TP*), correctly classified as a dissimilar pair (true negative, *TN*), incorrectly classified as similar pair while it is actually dissimilar (false positive, *FP*), and incorrectly classified as dissimilar pair while it is actually a similar pair (false negative, *FN*). Then, we create a confusion matrix for every tool containing the values of these *TP*, *FP*, *TN*, and *FN* frequencies. Subsequently the confusion matrix is used to compute an individual technique's performance.

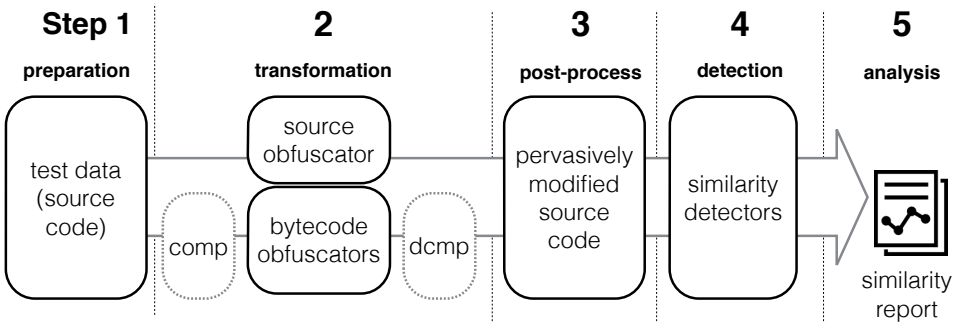

Fig. 1: The experimental framework

3.2 Tools and Techniques

Several tools and techniques were used in this study. These fall into three categories: obfuscators, decompilers, and detectors. The tool set included source and bytecode obfuscators, and two decompilers. The detectors cover a wide range of similarity measurement techniques and methods including plagiarism and clone detection, compression distance, string matching, and information retrieval. All tools are open source in order to expedite the repeatability of our experiments.

3.2.1 Obfuscators

In order to create pervasive modifications in Step 2 (transformation) of the framework, we used two obfuscators that do not employ strong obfuscations, Artifice and ProGuard. Artifice (Schulze and Meyer, 2013) is an Eclipse plugin for source-level obfuscation. The tool makes 5 different transformations to Java source code including 1) renaming of variables, fields, and methods, 2) changing assignment, increment, and decrement operations to normal form, 3) inserting additional assignment, increment, and decrement operations when possible, 4) changing `while` to `for` and the other way around, and 5) changing `if` to its short form. Artifice cannot be automated and has to be run manually because it is an Eclipse plugin. ProGuard (GuardSquare, 2015) is a well known open-source bytecode obfuscator. It is a versatile tool containing several functions including shrinking Java class files, optimisation, obfuscation, and pre-verification. ProGuard obfuscates Java bytecode by renaming classes, fields, and variables with short and meaningless ones. It also performs package hierarchy flattening, class repackaging, and modifying class and package access permissions.

3.2.2 Compiler and Decompilers

Our study uses compilation and decompilation for two purposes: transformation (obfuscation) and normalisation.

One can use a combination of compilation and decompilation as a method of source code obfuscation or transformation. Luo et al. (Luo et al, 2014) use GCC/G++ with different optimisation options to generate 10 different binary versions of the same program. However, if the desired final product is source code, a decompiler is also required in the process in order to transform the bytecode back to its source form.

Decompilation is a method for reversing the process of program compilation. Given a low-level language program such as an executable file, a decompiler generates a high-level language counterpart that resembles the (original) source code. This has several applications including recovery of lost source code, migrating a system to another platform, upgrading an old program into a newer programming language, restructuring poorly-written code, finding bugs or malicious code in binary programs, and program validation (Cifuentes and Gough, 1995). An example of using the decompiler to reuse code is a well-known lawsuit between Oracle and Google (United States District Court, 2011). It

seems that Google decompiled a Java library to obtain the source code of its APIs and then partially reused them in their Android operating system.

Since each decompiler has its own decompiling algorithm, one decompiler usually generates source code which is different from the source code generated by other decompilers. Using more than one decompiler can also be a method of obfuscation by creating variants of the same program with the same semantics but with different source code.

We selected two open source decompilers: Krakatau and Procyon. Krakatau (Grosse, 2016) is an open-source tool set comprising a decompiler, a classfile dissembler, and an assembler. Procyon (Strobel, 2016) includes a Java open-source decompiler. It has advantages over other decompilers for declaration of `enum`, `String`, `switch` statements, anonymous and named local classes, annotations, and method references. They are used in both the transformation (obfuscation) and normalisation post-process steps (Steps 2 and 3) of the framework.

The only compiler deployed in this study is the standard Java compiler (javac).

3.2.3 Plagiarism Detectors

The selected plagiarism detectors include JPlag, Sherlock, Sim, and Plaggie. JPlag (Prechelt et al, 2002) and Sim (Gitchell and Tran, 1999) are token-based tools which come in versions for text (jplag-text and simtext) and Java (jplag-java and simjava), while Sherock (Pike and Loki, 2002) relies on digital signatures (a number created from a series of bits converted from the source code text). Plaggie's detection (Ahtiainen et al, 2006) method is not public but claims to have the same functionalities as JPlag. Although there are several other plagiarism detection tools available, some of them could not be chosen for the study due to the absence of command-line versions preventing them from being automated. Moreover, we require a quantitative similarity measurement so we can compare their performances. All chosen tools report a numerical similarity value, $\text{sim}(x, y)$, for a given file pair x, y.

3.2.4 Clone Detectors

We cover a wide spectrum of clone detection techniques including text-based, token-based, and tree-based techniques. Like the plagiarism detectors, the selected tools are command-line based and produce clone reports providing a similarity value between two files.

Most state-of-the-art clone detectors do not report similarity values. Thus, we adopted the *General Clone Format (GCF)* as a common format for clone reports. We modified and integrated the *GCF Converter* (Wang et al, 2013) to convert clone reports generated by unsupported clone detectors into GCF format. Since a GCF report contains several clone fragments found between two files x and y, the similarity of x to y can be calculated as the ratio of the size of clone fragment found in x (overlaps are handled) to the size of x and vice versa.

$$\text{sim}_{\text{GCF}}(x, y) = \frac{\sum_{i=1}^{n} |frag_i(x)|}{|x|} \tag{2}$$

Using this method, we included five state-of-the-art clone detectors: CCFinderX, NICAD, Simian, iClones, and Deckard. CCFinderX (ccfx) (Kamiya et al, 2002) is a token-based clone detector detecting similarity using suffix trees. NICAD (Roy and Cordy, 2008) is a clone detection tool embedding TXL for pretty-printing, and compares source code using string similarity. Simian (Harris, 2015) is a pure, text-based, clone detection tool relying on text line comparison with a capability for checking basic code modifications, e.g. identifier renaming. iClones (Göde and Koschke, 2009) performs token-based incremental clone detection over several revisions of a program. Deckard (Jiang et al, 2007a) converts source code into an AST and computes similarity by comparing characteristic vectors generated from the AST to find cloned code based on approximate tree similarity.

Although most of clone reports only contain clone lines, the actual implementation of clone detection tools work at different granularity of code fragments. Measuring clone similarity at a single granularity level, such as line, may penalise some tools while favours another set of tools. With this concern in mind, our clone similarity calculation varies over multiple granularity levels to avoid biases to any particular tools. We consider three different granularity levels:

line, token, and character. Computing similarity at a level of lines or tokens is common for clone detectors. Simian and NICAD detect clones based on source code lines while CCFinderX and iClones work at token level. However, Deckard compares clones based on ASTs so its similarity does not come from neither lines nor tokens. To make sure that we get the most accurate similarity calculation for Deckard and other clone detectors, we also cover the most fine-grained level of source code: characters. With the three level of granularity (line, word, and character), we calculate three $\text{sim}_{\text{GCF}}(x, y)$ values for each of the tools.

3.2.5 Compression Tools

Normalised compression distance (NCD) is a distance metric between two documents based on compression (Cilibrasi and Vitányi, 2005). It is an approximation of the normalised information distance which is in turn based on the concept of Kolmogorov complexity (Li and Vitâanyi, 2008). The NCD between two documents can be computed by

$$\text{NCD}_z(x, y) = \frac{Z(xy) - \min\{Z(x), Z(y)\}}{\max\{Z(x), Z(y)\}} \tag{3}$$

where $Z(x)$ means the length of the compressed version of document x using compressor Z. In this study, five variations of NCD tools are chosen. One is part of CompLearn (Cilibrasi et al, 2015) which uses the built-in bzlib and zlib compressors. The other four have been created by the authors as shell scripts. The first one utilises 7-Zip (Pavlov, 2016) with various compression methods including BZip2, Deflate, Deflate64, PPMd, LZMA, and LZMA2. The other three rely on Linux's gzip, bzip2, and xz compressors respectively.

Lastly, we define another, asymmetric, similarity measurement based on compression called *inclusion compression divergence (ICD)*. It is a compressor based approximation to the ratio between the conditional Kolmogorov complexity of string x given string y and the Kolmogorov complexity of x, i.e. to $K(x|y)/K(x)$, the proportion of the randomness in x not due to that of y. It is defined as

$$\text{ICD}_Z(x, y) = \frac{Z(xy) - Z(y)}{Z(x)} \tag{4}$$

and when C is NCD_Z or ICD_Z then we use $\text{sim}_C(x, y) = 1 - C(x, y)$.

3.2.6 Other Techniques

We expanded our study with other techniques for measuring similarity including a range of libraries that measure textual similarity: difflib (Python Software Foundation, 2016) compares text sequences using Gestalt pattern matching, Python NGram (Poulter, 2012) compares text sequences via fuzzy search using n-grams, FuzzyWuzzy (Cohen, 2011) uses fuzzy string matching, jellyfish (Turk and Stephens, 2016) does approximate and phonetic matching of strings, and cosine similarity from scikit-learn (Pedregosa et al, 2011) which is a machine learning library providing data mining and data analysis. We also employed diff, the classic file comparison tool, and bsdiff, a binary file comparison tool. Using diff or bsdiff, we calculate the similarity between two Java files x and y using

$$\text{sim}_D(x, y) = 1 - \frac{\min(|y|, |D(x, y)|)}{|y|} \tag{5}$$

where $D(x, y)$ is the output of *diff* or *bsdiff*.

The result of $\text{sim}_D(x, y)$ is asymmetric as it depends on the size of the denominator. Hence $\text{sim}_D(x, y)$ usually produces a different result from $\text{sim}_D(y, x)$. This is because $\text{sim}_D(x, y)$ provides the distance of editing x into y which is different in the opposite direction.

The summary of all selected tools and their respective similarity measurement methods are presented in Table 1. The default configurations of each tools, as displayed in Table 2, are extracted from (1) the values displayed in the help menu of the tools, (2) the tools' websites, (3) or the tools' papers (e.g. Deckard (Jiang et al, 2007b)). The range of parameter values we searched for in our study are also included in Table 2.

Table 1: Tools with their similarity measures

Tool/Technique	Similarity calculation
Clone Det.	
ccfx (Kamiya et al, 2002)	tokens and suffix tree matching
deckard (Jiang et al, 2007b)	characteristic vectors of AST optimised by LSH
iclones (Göde and Koschke, 2009)	tokens and generalised suffix tree
nicad (Roy and Cordy, 2008)	TXL and string comparison (LCS)
simian (Harris, 2015)	line-based string comparison
Plagiarism Det.	
jplag-java (Prechelt et al, 2002)	tokens, Karp Rabin matching, Greedy String Tiling
jplag-text (Prechelt et al, 2002)	tokens, Karp Rabin matching, Greedy String Tiling
plaggie (Ahtiainen et al, 2006)	N/A (not disclosed)
sherlock (Pike and Loki, 2002)	digital signatures
simjava (Gitchell and Tran, 1999)	tokens and string alignment
simtext (Gitchell and Tran, 1999)	tokens and string alignment
Compression	
7zncd	NCD with 7z
bzip2ncd	NCD with bzip2
gzipncd	NCD with gzip
xz-ncd	NCD with xz
icd	Equation 4
ncd (Cilibrasi et al, 2015)	*ncd* tool with bzlib & zlib
Others	
bsdiff	Equation 5
diff	Equation 5
difflib (Python Software Foundation, 2016)	Gestalt pattern matching
fuzzywuzzy (Cohen, 2011)	fuzzy string matching
jellyfish (Turk and Stephens, 2016)	approximate and phonetic matching of strings
ngram (Poulter, 2012)	fuzzy search based using n-gram
sklearn (Pedregosa et al, 2011)	cosine similarity from machine learning library

4 Experimental Scenarios

To answer the research questions, four experimental scenarios have been designed and studied following the framework presented in Figure 1. The experiment was conducted on a virtual machine with 2.67 GHz CPU (dual core) and 2 GB RAM running Scientific Linux release 6.6 (Carbon), and 7 Microsoft Azure virtual machines with 4 cores, 14 GB RAM running Ubuntu 14.04 LTS. The details of each scenario are explained below.

4.1 Scenario 1 (Pervasive Modifications)

Scenario 1 studies tool performance against pervasive modifications (as simulated through source and bytecode obfuscation). At the same time, the best configuration for every tool is discovered. For this data set, we completed all the 5 steps of the framework: data preparation, transformation, post-processing, similarity detection, and analysing the similarity report. However, post-processing is limited to pretty printing and no normalisation through decompilation is applied.

4.1.1 Preparation, Transformation, and Normalisation

This section follows Steps 1 and 2 in the framework. The original data consists of 10 Java classes: *BubbleSort*, *EightQueens*, *GuessWord*, *TowerOfHanoi*, *InfixConverter*, *Kapreka_Transformation*, *MagicSquare*, *RailRoadCar*, *SLinkedList*, and *SqrtAlgorithm*. We downloaded them from two programming websites as shown in Table 3 along with the class descriptions. We selected only the classes that can be compiled and decompiled without any required dependencies other than the Java SDK. All of them are short Java programs with less than 200 LOC and they illustrate issues that are

Table 2: Tools and their parameters with chosen value ranges (DF denotes default parameters)

Tool	Settings	Details	DF	Range
Clone det.				
ccfx	b	min no. of tokens	50	3 4 5 10 15 16 17 18 19 20 21 22 23 24 25 30 35 40 45 50
	t	min token kinds	12	1 2 3 .. 14
deckard	mintoken	min no. of tokens	50	30, 50
	stride	sliding window size	inf	0, 1, 2, inf
	similarity	clone similarity	1.0	0.90, 0.95, 1.00
nicad	UPI	% of unique code	0.30	0.30, 0.50
	minline	min no. of lines	10	5, 8, 10
	rename	variable renaming	none	blind, consistent
	abstract	code abstraction	none	none, declaration, statement, expression, condition, literal
simian	threshold	min no. of lines	6	3 4 5 .. 10
	options	other options	none	none, ignoreCharacters, ignoreIdentifiers, ignoreLiterals, ignoreVariableNames
Plagiarism det.				
jplag-java	t	min no. of tokens	9	1 2 3 .. 12
jplag-text	t	min no. of tokens	9	1 2 3 .. 12
plaggie	M	min no. of tokens	11	1 2 3 .. 14
sherlock	N	chain length	4	1 2 3 .. 8
	Z	zero bits	3	0 1 2 .. 8
simjava	r	min run size	N/A	10 11 12 .. 24
simtext	r	min run size	N/A	4 5 6 .. 12
Compression				
7zncd-BZip2	mx	compression level	N/A	1 3 5 7 9
7zncd-Deflate	mx	compression level	N/A	1 3 5 7 9
7zncd-Deflate64	mx	compression level	N/A	1 3 5 7 9
7zncd-LZMA	mx	compression level	N/A	1 3 5 7 9
7zncd-LZMA2	mx	compression level	N/A	1 3 5 7 9
7zncd-PPMd	mx	compression level	N/A	1 3 5 7 9
bzip2ncd	C	block size	N/A	1 2 3 .. 9
gzipncd	C	compression speed	N/A	1 2 3 .. 9
icd	ma	compression algo.	N/A	BZip2, Deflate, Deflate64, LZMA, LZMA2, PPMd
	mx	compression level	N/A	1 3 5 7 9
ncd-zlib	N/A			
ncd-bzlib	N/A			
xzncd	-N	compression level	6	1 2 3 .. 9, e
Others				
bsdiff	N/A			
diff	N/A			
py-difflib	autojunk	auto. junk heuristic	N/A	true, false
	whitespace	ignoring white space	N/A	true, false
py-fuzzywuzzy	similarity	similarity calculation	N/A	ratio, partial_ratio, token_sort_ratio, token_set_ratio
py-jellyfish	distance	edit distance algo.	N/A	jaro_distance, jaro_winkler
py-ngram	N/A			
py-sklearn	N/A			

Table 3: Descriptions of the 10 original Java classes in the generated data set

No.	File	SLOC	Description
1	BubbleSort.java*	39	Bubble Sort implementation
2	EightQueens.java†	65	Solution to the Eight Queens problem
3	GuessWord.java*	115	A word guessing game
4	TowerOfHanoi.java*	141	The Tower of Hanoi game
5	InfixConverter.java*	95	Infix to postfix conversion
6	Kapreka_Transformation.java*	111	Kapreka Transformation of a number
7	MagicSquare.java†	121	Generating a Magic Square of size n
8	RailRoadCar.java*	71	Rearranging rail road cars
9	SLinkedList.java*	110	Singly linked list implementation
10	SqrtAlgorithm.java*	118	Calculating the square root of a number

* classes downloaded from http://www.softwareandfinance.com/Java

† classes downloaded from http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures

Table 4: Size of the data sets. The (generated) data set in Scenario 1 has been compiled and decompiled before performing the detection in Scenario 2 (generated*). The SOCO data set is used in Scenario 3.

Scenario	Data set	Files	#Comparisons	Positives	Negatives
1	generated	100	10,000	1,000	9,000
2	generated*	100	10,000	1,000	9,000
3	SOCO	259	67,081	453	66,628

usually discussed in basic programming classes. The process of test data preparation and transformation is illustrated in Figure 3. First, we selected each original source code file and obfuscated it using Artifice. This produced the first type of obfuscation: source-level obfuscation (No. 1). An example of a method before and after source-level obfuscation by Artifice is displayed on the top of Figure 2 (formatting has been adjusted due to space limits).

Next, both the original and the obfuscated versions were compiled to bytecode, producing two bytecode files. Then, both bytecode files were obfuscated once again by ProGuard, producing two more bytecode files.

All four bytecode files were then decompiled by either Krakatau or Procyon giving back eight additional obfuscated source code files. For example, No. 1 in Figure 3 is a pervasively modified version via source code obfuscation with Artifice. No. 2 is a version which is obfuscated by Artifice, compiled, obfuscated with ProGuard, and then decompiled with Krakatau. No. 3 is a version obfuscated by Artifice, compiled and then decompiled with Procyon. Using this method, we obtained 9 pervasively modified versions for each original source file, resulting in 100 files for the data set. The only post-processing step in this scenario is normalisation through pretty printing.

4.1.2 Similarity Detection

The generated data set of 100 Java code files is used for pairwise similarity detection in Step 4 of the framework in Figure 1, resulting in 10,000 pairs of source code files with their respective similarity values. We denote each pair and their similarity as a triple (x, y, sim). Since each tool can have multiple parameters to adjust and we aimed to cover as many parameter settings as possible, we repeatedly ran each tool several times with different settings in the range listed in Table 2. Hence, the number of reports generated by one tool equals the number of combinations of its parameter values. A tool with two parameters $p_1 \in P_1$ and $p_2 \in P_2$ has $|P1| \times |P2|$ different settings. For example, sherlock has two parameters $N \in \{1, 2, 3, ..., 8\}$ and $Z \in \{0, 1, 2, 3, ..., 8\}$. We needed to do $8 \times 9 \times 10,000 = 720,000$ pairwise

```
/* original */
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;
  ...
}
```

```
/* ARTIFICE */
public MagicSquare(int v2) {
  f00=new int[v2][v2];
  int v3;
  v3=0;
  while(v3<v2) {
    int v4;
    v4=0;
      while(v4<v2) {
        f00[v3][v4]=0;
        v4=v4+1;
      }
    v3=v3+1;
    ...
  }
```

```
/* original + Krakatau */
public MagicSquare(int i) {
  super();
  this.square=new int[i][i];
  int i0=0;
  int i1=0;
  while(i1<i) {
    this.square[i0][i1]=0;
    i1=i1+1;
  }
  i0=i0+1;
  ...
}
```

```
/* ARTIFICE + Krakatau */
public MagicSquare(int i) {
  super();
  this.f00=new int[i][i];
  int i0=0;
  int i1=0;
  while(i1<i){
    this.f00[i0][i1]=0;
    i1=i1+1;
  }
  i0=i0+1;
  ...
}
```

```
/* original + Procyon */
public MagicSquare(final int n) {
  super();
  this.square = new int[n][n];
  for (int i=0;i<n;++i) {
    for (int j=0;j<n;++j) {
      this.square[i][j]=0;
      }
  }
  ...
}
```

```
/* ARTIFICE + Procyon */
public MagicSquare(final int n) {
  super();
  this.f00=new int[n][n];
  for (int i=0;i<n;++i) {
    for (int j=0;j<n;++j) {
      this.f00[i][j]=0;
    }
  }
  ...
}
```

Fig. 2: The same code fragments, a constructor of `MagicSquare`, after pervasive modifications, and compilation/decompilation.

comparisons and generated 72 similarity reports. To cover the 30 tools with all of their possible configurations, we performed 14,880,000 pairwise comparisons in total and analysed 1,488 reports.

4.1.3 Analysing the Similarity Reports

In Step 5 of the framework, the results of the pairwise similarity detection are analysed. The 10,000 pairwise comparisons result in 10,000 (x, y, sim) entries. As in Equation 1, all pairs x, y are considered to be similar when the reported similarity sim is larger than a threshold T. Such a threshold must be set in an informed way to produce sensible results. However, as the results of our experiment will be extremely sensitive to the chosen threshold, we want to use the optimal threshold, i.e. the threshold that produces the best results. Therefore, we vary the cut-off threshold T between 0 and 100.

As shown in Table 4, the ground truth of the generated data set contains 1,000 positives and 9,000 negatives. The positive pairs are the pairs of files generated from the same original code. For example, all pairs that are the derivatives of `InfixConverter.java` must be reported as similar. The other 9,000 pairs are negatives since they come from different original source code files and must be classified as dissimilar. Using this ground truth, we can count the number of true and false positives in the results reported for each of the tools. We choose the F-score as the method to measure the tools'

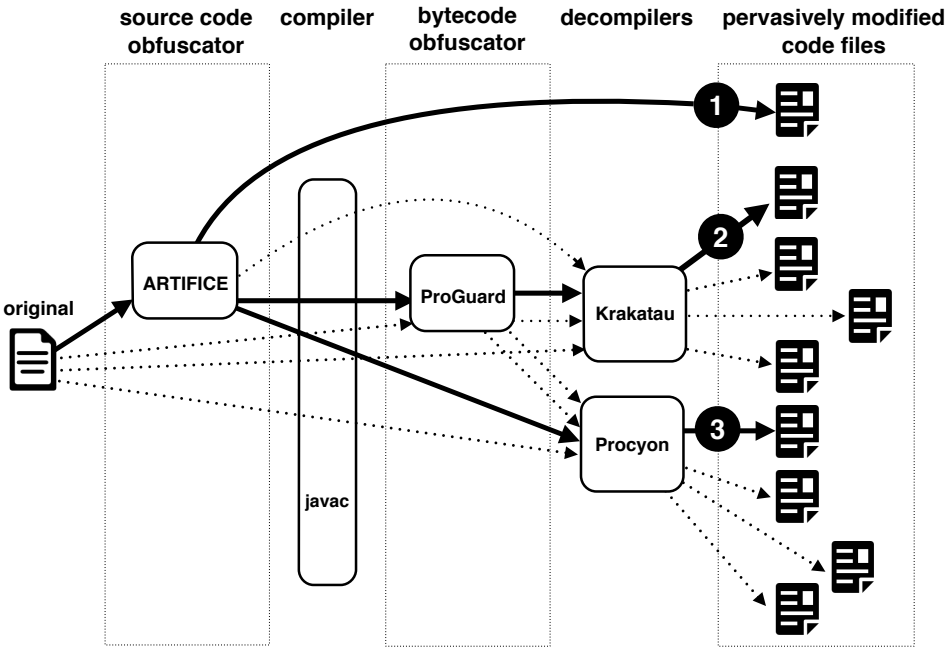

Fig. 3: Test data generation process

performance. The F-score is preferred in this context since the sets of similar files and dissimilar files are unbalanced and the F-score does not take true negatives into account[1].

The F-score is the harmonic mean of precision (ratio of correctly identified reused pairs to retrieved pairs) and recall (ratio of correctly identified pairs to all the identified pairs):

$$\text{precision} = \frac{TP}{TP + FP} \qquad \text{recall} = \frac{TP}{TP + FN}$$

$$\text{F-score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Using the F-score we can search for the best threshold T under which each tool has its optimal performance with the highest F-score. For example in Figure 4, after varying the threshold from 0 to 100, ncd-bzlib has the best threshold $T = 37$ with the highest F-score of 0.846. Since each tool may have more than one parameter setting, we call the combination of the parameter settings and threshold that produces the highest F-score the tool's "optimal configuration".

4.2 Scenario 2 (Decompilation)

We are interested in studying the effects of normalisation through compilation/decompilation before performing similarity detection. This is based on the observation that compilation has a normalising effect. Variable names disappear in bytecode and nominally different kinds of control structures can be replaced by the same bytecode, e.g. `for` and `while` loops are replaced by the same `if` and `goto` structures at bytecode level.

[1] For the same reason, we decided against using Matthews correlation coefficient (MCC).

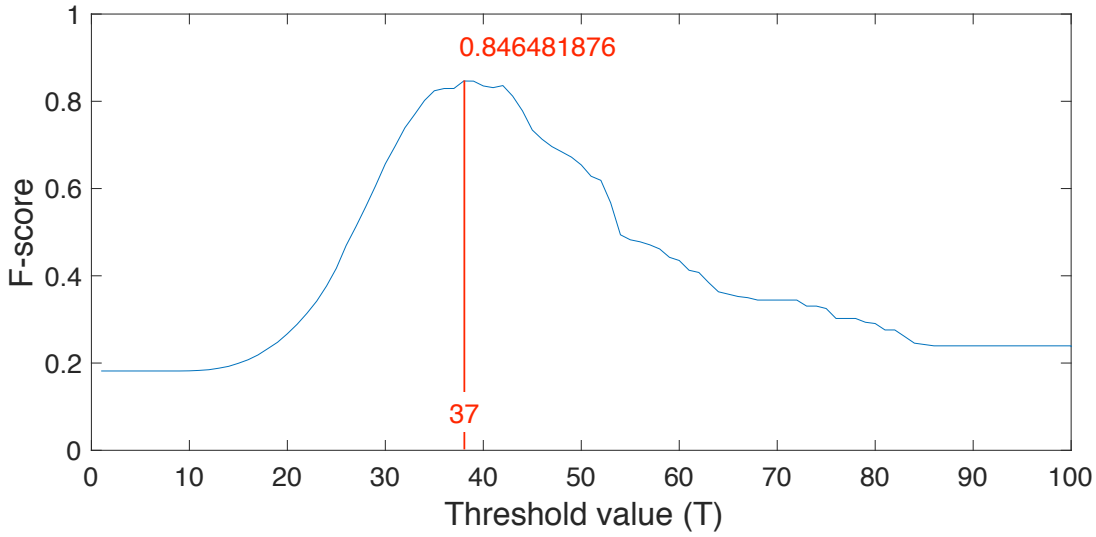

Fig. 4: The graph shows the F-score and the threshold values of ncd-bzlib. The tool reaches the highest F-score when the threshold equals 37.

Likewise, changes made by bytecode obfuscators may also be normalised by decompilers. Suppose a Java program P is obfuscated (transformed, T) into Q ($P \xrightarrow{T} Q$), then compiled (C) to bytecode B_Q, and decompiled (D) to source code Q' ($Q \xrightarrow{C} B_Q \xrightarrow{D} Q'$). This Q' should be different from both P and Q due to the changes caused by the compiler and decompiler. However, with the same original source code P, if it is compiled and decompiled using the same tools to create P' ($P \xrightarrow{C} B_P \xrightarrow{D} P'$), P' should have some similarity to Q' due to the analogous compiling/decompiling transformations made to both of them. Hence, one might apply similarity detection to find similarity sim(P', Q') and get more accurate results than sim(P, Q).

In this scenario, the data set is based on the same set of 100 source code files generated in Scenario 1. However, we added normalisation through decompilation to the post-processing (Step 3 in the framework) by compiling all the transformed files using javac and decompiling them using either Krakatau or Procyon. We then followed the same similarity detection and analysis process in Steps 4 and 5. The results are then compared to the results obtained from Scenario 1 to observe the effects of normalisation through decompilation.

4.3 Scenario 3 (Reused Boiler-Plate Code)

In this scenario, we analyse the tools' performance against an available data set that contains files in which fragments of boiler-plate code are reused with or without modifications. We choose the data set that has been provided by the Detection of SOurce COde Re-use competition for discovering monolingual re-used source code amongst a given set of programs (Flores et al, 2014), which we call the SOCO data set. We found that many of them share the same or very similar boiler-plate code fragments which perform the same task. Some of the boiler-plate fragments have been modified to adapt to the environment in which the fragments are re-used. Since we reused the data set from another study (Flores et al, 2014), we merely needed to format the source code files by removing comments and applying pretty-printing to them in step 1 of our experimental framework (see Figure 1). We later skipped step 2 and 3 of pervasive modifications and followed only step 4 – similarity detection, and step 5 – analysing similarity report in our framework.

We selected the Java training set containing 259 files for which the ground truth is provided. The ground truth contains 84 file pairs that share boiler-plate code. Using the provided pairs, we are able to measure both false positives and negatives. For each tool, this data set produced $259 \times 259 = 67,081$ pairwise comparisons. Out of these 67,081 file pairs, $259 + 2 \times 84 = 427$ pairs are similar. However, after manually investigating false positives in a preliminary study, we found that the provided ground truth contains errors. An investigation revealed that the provided ground truth contained two large clusters in which pairs were missing and that two given pairs were wrong[2]. After removing the wrong pairs and adding the missing pairs, the corrected ground truth contains 453 pairs.

We performed two analyses on this data set: 1) applying the derived configurations to the data set and measuring the tools' performances, and 2) searching for the optimal configurations. Again, no transformation or normalisation has been applied to this data set as it is already prepared. With a smaller subset of 10 tools, we performed 67,483,486 pairwise comparisons in total for this data set and analysed 1,006 similarity reports.

4.4 Scenario 4 (Ranked Results)

In our three previous scenarios, we compared the tools' performances using their optimal F-scores. The F-score offers weighted harmonic mean of precision and recall. It is a set-based measure that does not consider any ordering of results. The optimal F-scores are obtained by varying the threshold T to find the highest F-score. We observed from the results of the previous scenarios that the thresholds are highly sensitive to each particular data set. Therefore, we had to repeat the process of finding the optimal threshold every time we changed to a new data set. This was burdensome but could be done since we knew the ground truth data of the data sets. The configuration problem for clone detection tools including setting thresholds has been mentioned by several studies as one of the threats to validity (Wang et al, 2001). There has also been an initiative to avoid using thresholds at all for clone detection (Keivanloo et al, 2015).

Hence, we try to avoid the problem of threshold sensitivity affecting our results. Instead of looking at the results as a set and applying a cut-off threshold to obtain true and false positives, we consider only a subset of the results based on their rankings. We adopt three error measures mainly used in information retrieval: precision-at-n (prec@n), average r-precision (ARP), and mean average precision (MAP) to measure the tools' performances. We present their definitions below.

Given n as a number of top n results ranked by similarity, precision-at-n (Manning et al, 2009) is defined as:

$$\text{prec@}n = \frac{\text{TP}}{n}$$

In the presence of a ground truth, we can set the value of n to be the number of relevant results (i.e. true positives). With a know ground truth, precision-at-n when n equals to the number of true positives is called r-precision (RP) where r stands for "relevant" (Manning et al, 2009). If a set of relevant files for each query $q \in Q$ is $R_q = \{rf_{q_1}, ..., rf_{q_n}\}$, then the r-precisions for a query q is:

$$\text{RP}_q = \frac{\text{TP}_q}{|R_q|}$$

With presence of more than one query, an average r-precision (ARP) can be computed as the mean of all r-precision values (Beitzel et al, 2009):

$$\text{ARP} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \text{RP}_q$$

Lastly, mean average precision (MAP) measure the quality of results across several recall levels. It is calculated from multiple average precision-at-n values where n_{q_i} is the number of retrieved results after each relevant result $rf_{q_i} \in R_q$ of a query q is found. An average precision-at-n (aprec@n) of a query q is calculated from:

[2] The authors of the data set confirmed that the data set contains errors.

$$\text{aprec@n}_q = \frac{1}{|R_q|} \sum_{i=1}^{|R_q|} \text{prec@n}_{q_i}$$

Mean average precision (MAP) is then derived from the mean of all aprec@n values of all the queries in Q (Manning et al, 2009):

$$\text{MAP} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \text{aprec@n}_{q_i}$$

Precision-at-n, ARP, and MAP are used to measure how well the tools retrieve relevant results within top-n ranked items for a given query (Manning et al, 2009). We simulate a querying process by 1) running the tools on our data sets and generate similarity pairs, and 2) ranking the results based on their similarities reported by the tools. The higher similarity value, the higher the rank. The top ranked result has the highest similarity value. If a tie happens, we resort to ranking by alphabetical order of the file names.

For precision-at-n, the query is *"what are the most similar files in this data set?"* and we inspect only the top n results. Our calculation of precision-at-n in this study can be considered as a hybrid between a set-based and a ranked-based measure. We put the results from different original files in the same "set" and we "rank" them by their similarities. This is suitable for a case of plagiarism detection. To locate plagiarised source code files, one may not want to give a specific file as a query (since they do not know which file has been copied) but they want to retrieve a set of all similar pairs in a set ranked by their similarities. JPlag use this method to report plagiarised source code pairs (Prechelt et al, 2002).

For ARP and MAP, we calculate them by considering a query *"what are the most similar files for each given query q?"* For example, since we had a total of 100 files in our generated data set, we queried 100 times. We picked one file at at time from the data set as a query, and retrieved a ranked result of 100 files (including the query itself) according to the query. An r-precision was calculated from the top 10 results. We limited at only the top 10 results since our ground truth contain 10 pervasively modified versions for each original source code file (including itself). Thus, the number of relevant results, r, is 10 in this study. We derive ARP from the average of the 100 r-precision values. The same process is repeated for MAP except using average precision-at-n instead of r-precision.

Using these three error measures, we can compare the performance of similarity detection techniques and tools by not relying on the threshold at all. They also provide another aspect of evaluating the tools performances. We can observe how well the tools report correct results within the top n pairs.

5 Results

We used the four experimental scenarios of pervasive modifications, decompilation, reused boiler-plate code, and ranked results to answer the five research questions. The execution of 30 similarity analysers on the data sets along with searching for their optimal parameters took several months to complete. We carefully observed and analysed the similarity reports and the results are discussed below according to the order of the five research questions.

5.1 RQ1: Performance Comparison

How well do current similarity detection techniques perform in the presence of pervasive source code modifications?

The summary of the tools' performances and their optimal configurations on the generated data set are listed in Table 5. We show seven error measures in the table including false positives (FP), false negatives (FN), accuracy (Acc), precision (Prec), recall (Rec), area under ROC curve (AUC), and F-score (F1). The tools are classified into 4 groups: clone detection tools, plagiarism detection tools, compression tools, and other similarity analysers. We can see that the

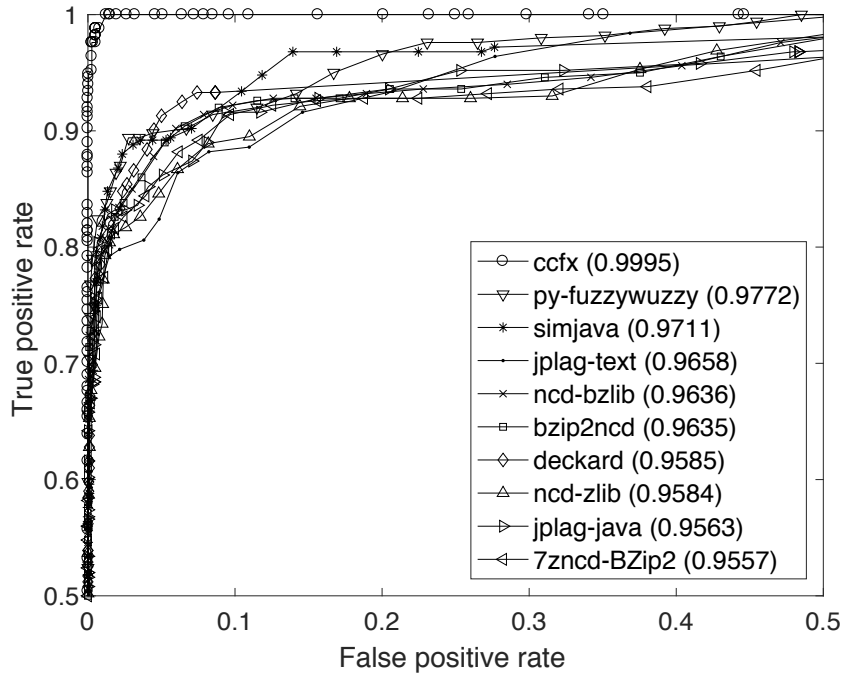

Fig. 5: The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC).

tools' performances vary over the same data set. For clone detectors, we applied three different granularity levels of similarity calculation: line (L), token (T), and character (C). We find that measuring code similarity at different code granularity levels impacts the performance of the tools. For example, ccfx gives the higher F-score when measuring similarity at character level than at line or token level. We present only the results of the best granularity level here. The complete results of the tools can be downloaded from the study website[3], including the generated data set before and after compilation/decompilation.

In terms of accuracy and F-score, the token-based clone detector ccfx is the winner. The top 10 tools with highest F-score include ccfx (0.9760) followed by py-fuzzywuzzy (0.876), jplag-java (0.8636), py-difflib (0.8629), simjava (0.0.8618), deckard (0.8509), bzip2ncd (0.8494), ncd-bzlib (0.8465), simian (0.8413), and ncd-zlib (0.8361) respectively. Interestingly, tools from all the four groups appear in the top ten.

For clone detectors, we have a token-based tool (ccfx), an AST-based tool (deckard), and a string-based tool (simian) in the top ten. This shows that with pervasive modifications, multiple clone detectors with different detection techniques can offer comparable results given their optimal configurations are provided. However, some clone detectors, e.g. iclones and nicad, did not perform well in this data set.

For plagiarism detection tools, jplag-java and simjava, which are token-based plagiarism detectors, are the leaders. Other plagiarism detectors give acceptable performance except simtext. This is not a surprise since the tool is intended for using with natural text rather than source code.

Compression tools show promising results of NCD for code similarity detection. They are ranked mostly in the middle from 7th to 24th with comparable results. The three bzip2-based NCD implementations, ncd-zlib, ncd-bzlib, and bzip2ncd only slightly outperform other compressors like gzip or LZMA. So the actual compression method may not have a strong effect in this context.

[3] http://crest.cs.ucl.ac.uk/resources/cloplag/

Other techniques for code similarity offer varied performance. Tools such as diff, bsdiff, ngram and sklearn perform badly. They are ranked among the last positions at 30th, 26th, 22th, and 28th respectively. Surprisingly, two Python tools using difflib and fuzzywuzzy string matching techniques produce very high F-scores.

To find the overall performance over similarity thresholds of 0 to 100, we drew the receiver operating characteristic (ROC) curves, calculated the area under the curve (AUC), and compared them. The closer the value is to one, the better the tool's performance. Figure 5 include the ten highest AUC tools. We can see from the figure that ccfx is again the winner with the highest AUC (0.9995), followed by py-fuzzywuzzy (0.9772), simjava (0.9711), jplag-text (0.9658), ncd-bzlib (0.9636), bzip2ncd (0.9635), deckard (0.9585), and ncd-zlib (0.9584). The two other tools, jplag-java and 7zncd-BZip2, offer AUCs of 0.9563 and 0.9557.

The best tool with respect to accuracy, and F-score is ccfx. The tool with the lowest false positive is py-difflib. The lowest false negatives is given by diff. However, considering the large amount of false positive for diff (8,810 false postives which mean 8,810 out of 9,000 dissimilar files are treated as similar), the tool tends to judge everything as similar. The second lowest false negative is once again ccfx.

Compared to our previous study (Ragkhitwetsagul et al, 2016), we similarly found that specialised tools such as source code clone and plagiarism detectors perform well against pervasively modified code. They were better than most of the compression-based and general string similarity tools. Compression-based tools mostly give decent and comparable results for all compression algorithms. String similarity tools perform poorly and mostly ranked among the lasts. However, we found that Python difflib and fuzzywuzzy perform surprisingly better with this expanded version of the data set than the original data set. They are both now ranked highly among the top 5.

RQ2: Optimal Configurations

What are the best parameter settings and similarity thresholds for the techniques?

We thoroughly analysed various configurations of every tool and found that some specific settings are sensitive to pervasively modified code while others are not. The complete list of the best configurations of every tool from Scenario 1 can be found in the second column of Table 5. The optimal configurations are significantly different from the default configurations, in particular for the clone detectors. For example, using the default settings for ccfx (b=50, t=12) leads to a very low F-score of 0.5781 due to a very high number of false negatives. Interestingly, a previous study on agreement of clone detectors (Wang et al, 2013) observed the same difference between default and optimal configurations.

In addition, we performed a detailed analysis of ccfx's configurations. This is because ccfx is a widely-used tool in several clone research studies. Two parameter settings are chosen for ccfx in this study: b, the minimum length of clone fragments in the unit of tokens, and t, the minimum number of kinds of tokens in clone fragments. We initially observed that the optimal F-scores of the tool were at either b=5 or b=19. Hence, we expanded the search space of ccfx parameters from 280 ($|b| = 20 \times |t| = 14$) to 392 settings ($|b| = 28 \times |t| = 14$) to make sure that we did not find a local optimum. We did a fine-grained search of b starting from 3 to 25 stepping by one and coarse-grained search from 30 to 50 stepping by 5.

From Figure 6, we can see that the default settings of ccfx, b=50 and t=12 (denoted with a $\times$ symbol), provides a decent precision but very low recall. We observed that one cannot tune ccfx to obtain the highest precision without sacrificing recall. The best settings for precision and recall of ccfx are described in Table 6. The ccfx tool gives the best precision with b=19 and t=7, 8, 9 and gives the best recall with b=5 and t=12.

The landscape of ccfx performance in terms of F-score is depicted in Figure 7. Visually, we can distinguish regions that are the sweet spot for ccfx's parameter settings against pervasive modifications from the rest. There are two regions covering the b value of 19 with t value from 7 to 9, and b value of 5 with t value from 11 to 12. The two regions provide F-scores ranging from 0.9589 up to 0.9760.

Table 5: Optimal configuration of every tool and technique obtained from the generated data set in Scenario 1 and their rankings (**R**) by F-scores (**F1**).

Tool	**Settings**	***T***	FP	FN	Acc	Prec	Rec	AUC	**F1**	**R**
Clone det.										
ccfx (C)*	b=5,t=11	36	24	24	0.9952	0.9760	0.9760	0.9995	0.9760	1
deckard (T)*	mintoken=30 stride=2 similarity=0.95	17	44	227	0.9729	0.9461	0.7730	0.9585	0.8509	6
iclones (L)*	minblock=10 minclone=50	0	36	358	0.9196	0.9048	0.4886	0.7088	0.6345	27
nicad (L)*	UPI=0.50 minline=8 rename=blind abstract=literal	38	38	346	0.9616	0.9451	0.6540	0.8164	0.7730	23
simian (C)*	threshold=4 ignoreVariableNames	5	150	165	0.9685	0.8477	0.8350	0.9262	0.8413	9
Plagiarism det.										
jplag-java	t=7	19	58	196	0.9746	0.9327	0.8040	0.9563	0.8636	3
jplag-text	t=4	14	66	239	0.9695	0.9202	0.7610	0.9658	0.8331	12
plaggie	M=8	19	83	234	0.9683	0.9022	0.7660	0.9546	0.8286	15
sherlock	N=4, Z=2	6	142	196	0.9662	0.8499	0.8040	0.9447	0.8263	17
simjava	r=16	15	120	152	0.9728	0.8760	0.8480	0.9711	0.8618	5
simtext	r=4	14	38	422	0.9540	0.9383	0.5780	0.8075	0.7153	25
Compression										
7zncd-BZip2	mx=1,3,5	45	64	244	0.9692	0.9220	0.7560	0.9557	0.8308	14
7zncd-Deflate	mx=7	38	122	215	0.9663	0.8655	0.7850	0.9454	0.8233	20
7zncd-Deflate64	mx=7,9	38	123	215	0.9662	0.8645	0.7850	0.9453	0.8229	21
7zncd-LZMA	mx=7,9	41	115	213	0.9672	0.8725	0.7870	0.9483	0.8275	16
7zncd-LZMA2	mx=7,9	41	118	213	0.9669	0.8696	0.7870	0.9482	0.8262	18
7zncd-PPMd	mx=9	42	140	198	0.9662	0.8514	0.8020	0.9467	0.8260	19
bzip2ncd	C=1..9	38	62	216	0.9722	0.9267	0.7840	0.9635	0.8494	7
gzipncd	C=7	31	110	203	0.9687	0.8787	0.7970	0.9556	0.8359	11
icd	ma=LZMA2 mx=7,9	50	86	356	0.9558	0.8822	0.6440	0.9265	0.7445	24
ncd-zlib	N/A	30	104	207	0.9689	0.8841	0.7930	0.9584	0.8361	10
ncd-bzlib	N/A	37	82	206	0.9712	0.9064	0.7940	0.9636	0.8465	8
xzncd	-e	39	120	203	0.9677	0.8691	0.7970	0.9516	0.8315	13
Others										
bsdiff*	N/A	71	199	577	0.9224	0.6801	0.4230	0.8562	0.5216	30
diff (C)*	N/A	8	626	184	0.9190	0.5659	0.8160	0.9364	0.6683	26
py-difflib	whitespace=false autojunk=false	28	12	232	0.9756	0.9846	0.7680	0.9412	0.8629	4
py-fuzzywuzzy	token_set_ratio	85	58	176	0.9766	0.9342	0.8240	0.9772	0.8757	2
py-jellyfish	jaro_distance	78	340	478	0.9182	0.6056	0.5220	0.8619	0.5607	29
py-ngram	N/A	49	110	224	0.9666	0.8758	0.7760	0.9410	0.8229	22
py-sklearn	N/A	48	292	458	0.9250	0.6499	0.5420	0.9113	0.5911	28

* Tools that do not report similarity value directly. Similarity is measured at the granularity level of line (L), token (T), or character (C).

Table 6: ccfx's parameter settings for the highest precision and recall

Error measure	Value	ccfx's parameters	
		b	*t*
Precision	1.000	19	7 8 9
Recall	0.980	5	12

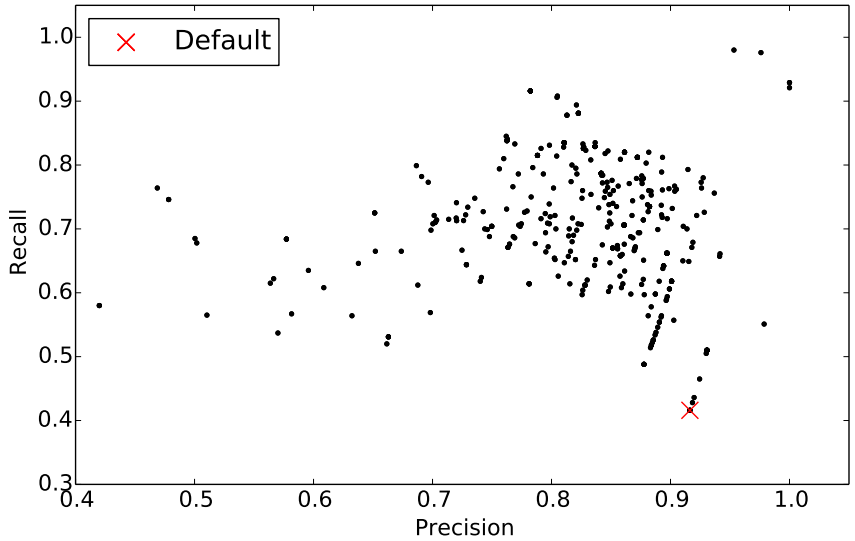

Fig. 6: Trade off between precision and recall of 392 ccfx parameter settings. The default settings provides low precision and recall.

RQ3: Normalisation by Decompilation

How much does compilation followed by decompilation as a pre-processing normalisation method improve detection results?

The results after adding compilation and decompilation for normalisation to the post-processing step before performing similarity detection is shown in Figure 8. We can clearly observe that decompilation by both Krakatau and Procyon boosts the F-scores of every tool in the study.

Table 7 shows the performances of the tools after decompilation by Krakatau in terms of false positive (FP), false negative (FN), accuracy (Acc), precision (Prec), recall (Rec), area under ROC curve (AUC), and F-score. We can see that normalisation by compilation/decompilation has a strong effect on the number of false results reported by the tools. Every tool has the amount of false positives and negatives greatly reduced and three tools, simian, jplag-java, and simjava, even no longer report any false result. All compression or other techniques still report some false results.

To confirm this, we carefully investigated the source code after normalisation and found that decompiled files created by Krakatau are very similar despite the applied obfuscation. As depicted in Figure 2 in the middle, the two code fragments become very similar after compilation and decompilation by Krakatau. This is because Krakatau has been designed to be robust to minor obfuscations and the transformations made by Artifice and ProGuard are not very complex. Code normalisation by Krakatau resulted in multiple optimal configurations found for some of the tools. We selected only one optimal configuration to include in Table 7 and separately reported the complete list of optimal configurations in Table 8.

Table 7: Optimal configuration of every tool obtained from the generated* data set decompiled by Krakatau in Scenario 2 and their rankings (**R**) by F-scores (**F1**).

Tool	**Settings**	***T***	FP	FN	Acc	Prec	Rec	AUC	**F1**	**R**
Clone det.										
ccfx*† (T)	b=5, t=8	50	0	18	0.9982	1.0000	0.9820	0.9991	0.9909	4
deckard*† (L)	mintoken=30 stride=1 similarity=0.95	29	0	84	0.9916	1.0000	0.9160	0.9459	0.9562	11
iclones* (L)	minblock=8 minclone=50	10	0	86	0.9914	1.0000	0.9140	0.9610	0.9551	14
nicad*† (T)	UPI=0.30 minline=8 rename=blind abstract=literal	19	0	106	0.9894	1.0000	0.8940	0.9526	0.9440	24
simian*† (T)	threshold=3 ignoreidentifiers	17	0	0	1.0000	1.0000	1.0000	0.9960	1.0000	1
Plagiarism det.										
jplag-java	t=4..12,default	23	0	0	1.0000	1.0000	1.0000	0.9964	1.0000	1
jplag-text	t=1	56	16	24	0.9960	0.9839	0.9760	0.9993	0.9799	6
plaggie	M=9	29	0	84	0.9916	1.0000	0.9160	0.9454	0.9562	13
sherlock	N=1,Z=0	60	34	22	0.9944	0.9664	0.9780	0.9989	0.9722	7
simjava†	r=18	17	0	0	1.0000	1.0000	1.0000	0.9998	1.0000	1
simtext	r=4; r=5	33 31	33	60	0.9907	0.9661	0.9400	0.9862	0.9529	16
Compression										
7zncd-BZip2	mx=1,3,5	49	40	40	0.9920	0.9600	0.9600	0.9983	0.9600	10
7zncd-Deflate	mx=9	46	28	71	0.9901	0.9707	0.9290	0.9978	0.9494	18
7zncd-Deflate64	mx=9	46	28	72	0.9900	0.9707	0.9280	0.9978	0.9489	19
7zncd-LZMA	mx=7,9	48	28	72	0.9900	0.9707	0.9280	0.9977	0.9489	19
7zncd-LZMA2	mx=7,9	48	28	72	0.9900	0.9707	0.9280	0.9977	0.9489	19
7zncd-PPMd	mx=9	49	40	31	0.9929	0.9604	0.9690	0.9985	0.9647	8
bzip2ncd	C=1..9,default	43	40	36	0.9924	0.9602	0.9640	0.9983	0.9621	9
gzipncd	C=8,9	38	28	63	0.9909	0.9710	0.9370	0.9980	0.9537	15
icd†	ma=LZMA, mx=7,9	54	45	68	0.9887	0.9539	0.9320	0.9921	0.9428	25
ncd-zlib	N/A	37	28	72	0.9900	0.9707	0.9280	0.9981	0.9489	19
ncd-bzlib	N/A	42	46	36	0.9918	0.9545	0.9640	0.9984	0.9592	11
xzncd	-1	43	16	83	0.9901	0.9829	0.9170	0.9967	0.9488	23
Others										
bsdiff	N/A	78	0	171	0.9829	1.0000	0.8290	0.9595	0.9065	28
diff (C)	N/A	23	12	186	0.9802	0.9855	0.8140	0.9768	0.8916	29
py-difflib	autojunk=true	23	28	66	0.9906	0.9709	0.9340	0.9823	0.9521	17
py-fuzzywuzzy	token_set_ratio	90	0	32	0.9968	1.0000	0.9680	0.9966	0.9837	5
py-jellyfish	jaro_winkler	89	40	220	0.9740	0.9512	0.7800	0.9473	0.8571	30
py-ngram	N/A	60	48	104	0.9848	0.9492	0.8960	0.9726	0.9218	26
py-sklearn	N/A	68	98	66	0.9836	0.9050	0.9340	0.9955	0.9193	27

* Tools that do not report similarity value directly. The similarity is measured at the granularity level of line (L), token (T), or character (C).
† Tools that have several optimal configurations. The complete lists can be found in Table 8.

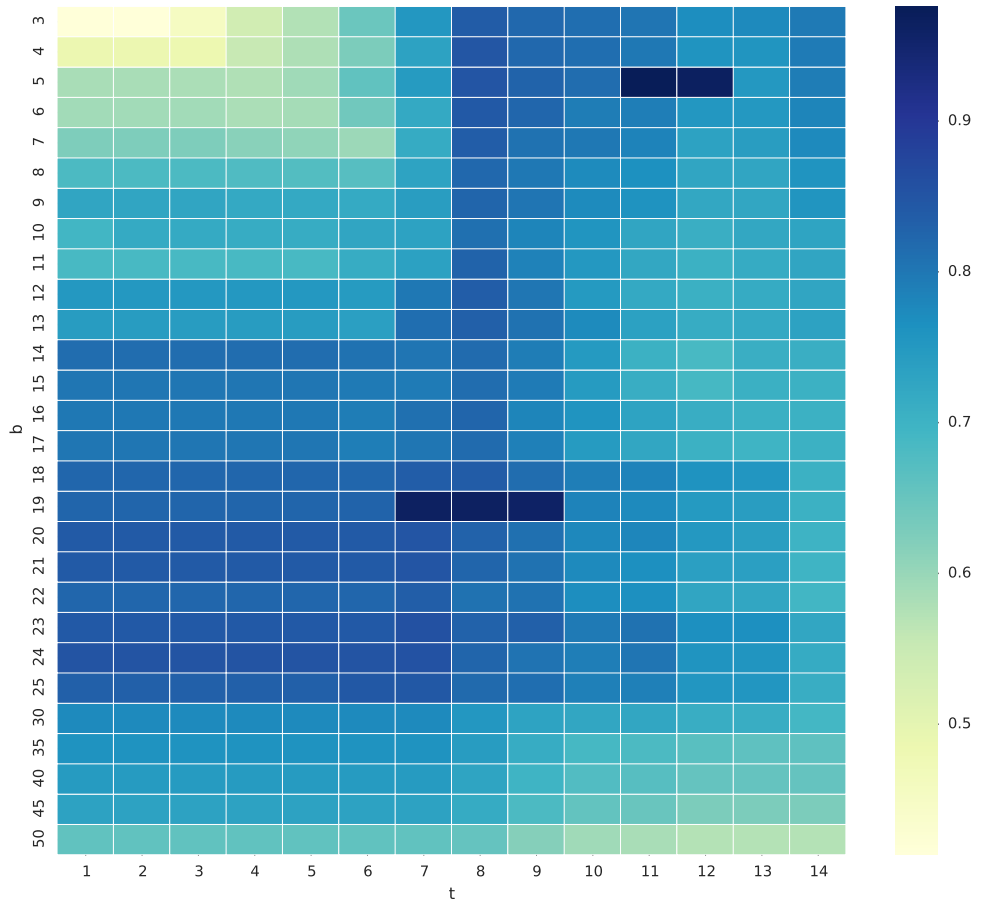

Fig. 7: F-scores of 392 ccfx's b and t parameter values

Normalisation via decompilation with Procyon also improves the performance of the similarity detectors, but not as much as Krakatau (see Table 9). Interestingly, Procyon performs slightly better for deckard, sherlock, and py-sklearn. An example of code before and after decompilation by Procyon is shown in Figure 2 at the bottom.

The main difference between Krakatau and Procyon is that Procyon attempts to produce much more high-level source code while Krakatau's is nearer to the bytecode. It seems that the low-level approach of Krakatau has a stronger normalisation effect. Hence, compilation/decompilation may be used as an effective normalisation method that greatly improves similarity detection between Java source code.

5.2 RQ4: Reuse of Configurations

Can we reuse optimal configurations from one data set in another data set effectively?

For the 10 highest ranking tools from RQ1, we applied the derived optimal configurations obtained from the generated data set (denoted as $C_{\text{generated}}$) to the SOCO data set. Table 11 shows that using these configurations has a detrimental impact on the similarity detection results for another data set, even for tools that have no parameters (e.g. ncd-zlib and ncd-bzlib) and are only influenced by the chosen similarity threshold.

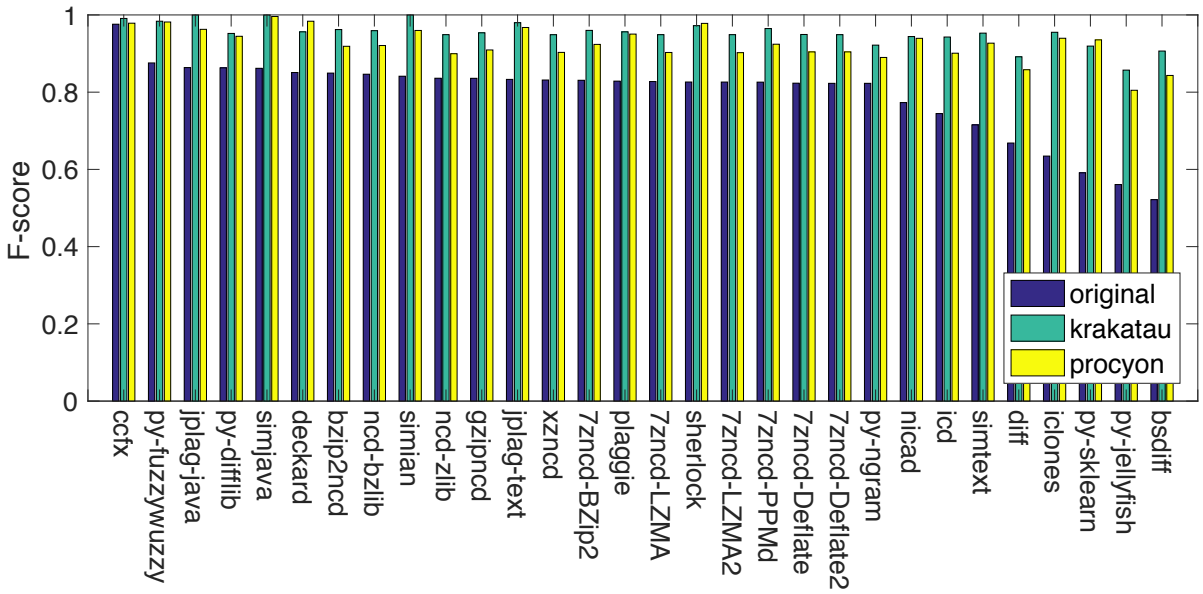

Fig. 8: Comparison of tool performances (F-score) before and after decompilation

Table 8: Complete set of optimal configurations for ccfx, deckard, simian, simjava, and icd obtained from the generated* data set decompiled by Krakatau in Scenario 2

Tool	**Settings**	**Granularity***	***T***
ccfx	b=5,t=8;	T	50
	b=5,t=11	T	17
deckard	mintoken=30, stride=1, similarity=0.95	L, T, C	29,29,34
	mintoken=30, stride=1, similarity=1.00	L, T, C	10,12,15
	mintoken=30, stride=2, similarity=0.90	T	54
	mintoken=30, stride=2, similarity=0.95	L, T, C	22,28,32
	mintoken=30, stride=inf, similarity=0.95	L, T, C	29,29,34
	mintoken=30, stride=inf, similarity=1.00	L, T, C	10,12,15
	mintoken=50, stride=1, similarity=0.95	L, T, C	23,29,31
	mintoken=50, stride=2, similarity=0.95	L, T, C	21,18,23
	mintoken=50, stride=inf, similarity=0.95	L, T, C	23,28,31
simian	threshold={3,4}, ignoreidentifiers	T	17
	threshold=3, ignoreidentifiers	C	
	threshold=5, ignoreidentifiers	L	12
	threshold=5, ignoreidentifiers	T	13
simjava	r=18,19;		17
	r=20;		16
	r=26,27;		11
	r=28;		9
	r=default		27
simtext	r=4;		33
	r=5		31
icd	ma=LZMA, mx=7,9;		54
	ma=LZMA2, mx=7,9		

* Similarity from GCF clone report is calculated at three different granularity levels: L=line, T=token, C=character.

Table 9: Optimal configuration of every tool obtained from the generated* data set (decompiled by Procyon) in Scenario 2 and their rankings (**R**) by F-scores (**F1**).

Tool	**Settings**	***T***	FP	FN	Acc	Prec	Rec	AUC	**F1**	**R**
Clone det.										
ccfx* (L)	b=20, t=1..7	11	4	38	0.9958	0.9959	0.962	0.9970	0.9786	4
deckard* (T)	mintoken=30 stride=1, inf similarity=1.00	10	0	32	0.9968	1.0000	0.9680	0.9978	0.9837	2
iclones* (C)	minblock=10 minclone=50	0	18	98	0.9884	0.9804	0.9020	0.9508	0.9396	11
nicad* (W)	UPI=0.30 minline=10 rename=blind abstract= condition,literal	11	16	100	0.9884	0.9825	0.9000	0.9536	0.9394	12
simian* (C)	threshold=3 ignoreIdentifiers	23	8	70	0.9922	0.9915	0.9300	0.9987	0.9598	8
Plagiarism det.										
jplag-java	t=8	22	0	72	0.9928	1.0000	0.9280	0.9887	0.9627	7
jplag-text	t=9	11	16	48	0.9936	0.9835	0.9520	0.9982	0.9675	6
plaggie	M=13,14	10	16	80	0.9904	0.9829	0.9200	0.9773	0.9504	9
sherlock	N=1, Z=0	55	28	16	0.9956	0.9723	0.9840	0.9997	0.9781	5
simjava	r=default	11	8	0	0.9992	0.9921	1.0000	0.9999	0.9960	1
simtext	r=4	15	42	100	0.9858	0.9554	0.9000	0.9686	0.9269	14
	r=default	0								
Compression										
7zncd-BZip2	mx=1,3,5	51	30	116	0.9854	0.9672	0.8840	0.9909	0.9237	16
7zncd-Deflate	mx=9	49	25	154	0.9821	0.9713	0.8460	0.9827	0.9043	20
7zncd-Deflate64	mx=9	49	25	154	0.9821	0.9713	0.8460	0.9827	0.9043	20
7zncd-LZMA	mx=7,9	52	16	164	0.9820	0.9812	0.8360	0.9843	0.9028	23
7zncd-LZMA2	mx=7,9	52	17	164	0.9819	0.9801	0.8360	0.9841	0.9023	24
7zncd-PPMd	mx=9	53	22	122	0.9856	0.9756	0.8780	0.9861	0.9242	15
bzip2ncd	C=1..9,default	47	12	140	0.9848	0.9862	0.8600	0.9922	0.9188	18
gzipncd	C=3	36	40	133	0.9827	0.9559	0.8670	0.9846	0.9093	25
icd	ma=LZMA, mx=7,9 ma=LZMA2, mx=7,9	54	37	150	0.9813	0.9583	0.8500	0.9721	0.9009	
ncd-zlib	N/A	41	30	158	0.9812	0.9656	0.8420	0.9876	0.8996	26
ncd-bzlib	N/A	47	8	140	0.9852	0.9908	0.8600	0.9923	0.9208	17
xzncd	-e	49	35	148	0.9817	0.9605	0.8520	0.9860	0.9030	22
Others										
bsdiff	N/A	73	48	236	0.9716	0.9409	0.7640	0.9606	0.8433	29
diff (C)	N/A	23	6	244	0.9750	0.9921	0.7560	0.9826	0.8581	28
py-difflib	autojunk=true	26	12	94	0.9894	0.9869	0.9060	0.9788	0.9447	10
py-fuzzywuzzy	token_set_ratio	90	0	36	0.9964	1.0000	0.9640	0.9992	0.9817	3
py-jellyfish	jaro_winkler	87	84	270	0.9646	0.8968	0.7300	0.9218	0.8049	30
py-ngram	N/A	58	8	192	0.9800	0.9902	0.8080	0.9714	0.8899	27
py-sklearn	N/A	69	54	74	0.9872	0.9449	0.9260	0.9897	0.9354	12

* Tools that do not report similarity value directly. The similarity is measured at the granularity level of line (L), token (T), or character (C).

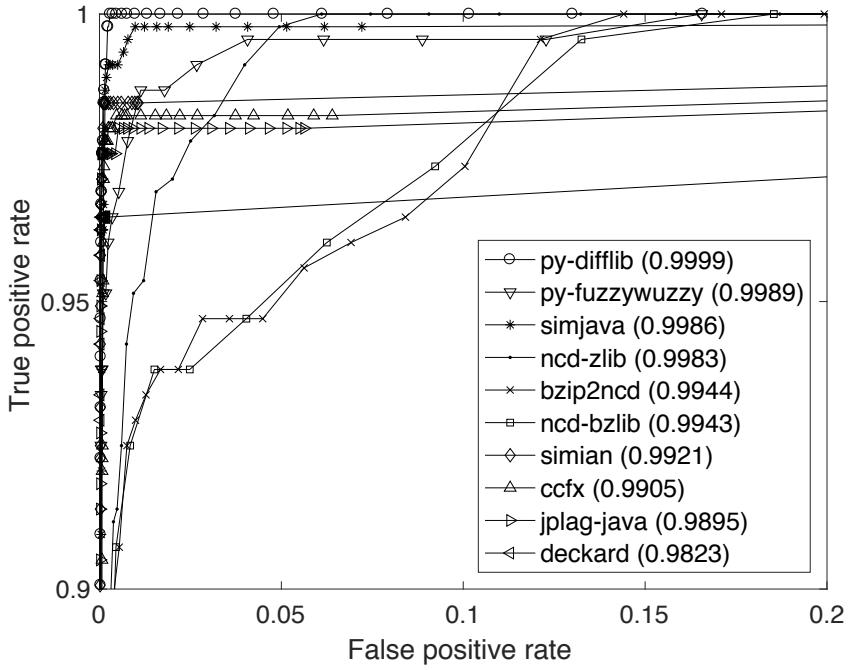

Fig. 9: The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC) for SOCO.

To confirm this, we again searched for the best configurations (settings and threshold) for the SOCO data set, which are listed in Table 11. The reported F-scores are very high for the dataset-based optimal configurations (denoted as C_{soco}), confirming that configurations are very sensitive to the data set on which the similarity detection is applied. We found the dataset-based optimal configurations, C_{soco}, to be very different from the configuration for the generated data set $C_{\text{generated}}$. We report the complete evaluation of the tools on the SOCO data set with the optimal configurations in Table 10. Among the 10 tools, the winner in terms of F-score is simian (0.9593), followed by jplag-java (0.9576) and deckard (0.9520). Regarding the AUC under ROC curve, the results are illustrated in Figure 9. The tool with the highest AUC is py-difflib (0.9999).

Lastly, we noticed that the best thresholds of the tools are very different from one data set to another and that the chosen similarity threshold tends to have the largest impact on the performance of similarity detection. This observation provides further motivation for a threshold-free comparison using precision-at-n.

5.3 RQ5: Ranked Results

How accurate are the results when only the top n results are retrieved?

We applied additional three error measures; precision-at-n (prec@n), average *r*-precision (ARP) and mean average precision (MAP); adopted from information retrieval to the generated and SOCO data set. The results are discussed below.

Precision-at-n: As discussed in Section 4.4, we used prec@n in a pair-based manner. For the generated data set, we sorted the 10,000 pairs of documents by their similarity values from the highest to the lowest. Then, we evaluated the tools based on a set of top *n* elements. We varied the value of *n* from 100 to 1500. In Table 12, we only reported the *n* equals to 1,000 since it is the number of true positives in the data set. The ccfx tool has the highest prec@n of 0.9760 (it also has F-score of 0.9760) followed by simjava (0.8600), and py-fuzzywuzzy (0.8570). Compared to F-scores, the

Table 10: Optimal configuration of every tool and technique obtained from the SOCO data set and their rankings (**R**) by F-scores (**F1**).

Tools	**Settings**	***T***	FP	FN	Acc	Prec	Rec	AUC	**F1**	**R**
ccfx (C)*	b={15 16 17} t=12	25	42	15	0.9992	0.9125	0.9669	0.9905	0.9389	5
py-fuzzywuzzy	ratio	65	30	30	0.9991	0.9338	0.9338	0.9989	0.9338	6
jplag-java	t=12	29	26	13	0.9994	0.9442	0.9713	0.9895	0.9576	2
py-difflib	autojunk=true whitespace=true	42	30	21	0.9992	0.9351	0.9536	0.9999	0.9443	4
simjava	r=25	23	40	28	0.9990	0.9140	0.9382	0.9986	0.9259	7
deckard (T)*	mintoken=50 stride=2 similarity=1.00	19	27	17	0.9993	0.9417	0.9625	0.9823	0.9520	3
bzip2ncd	C=1 2 3 .. 8 9	54	20	94	0.9983	0.9473	0.7925	0.9944	0.8630	10
ncd-bzlib	N/A	52	30	82	0.9983	0.9252	0.8190	0.9943	0.8689	9
simian (L)*	threshold=4 ignore VariableNames	26	20	17	0.9994	0.9561	0.9625	0.9921	0.9593	1
ncd-zlib	N/A	57	10	91	0.9985	0.9731	0.7991	0.9983	0.8776	8

* Tools that do not report similarity value directly.

Table 11: The table displays the results after applying the best configurations ($C_{\text{generated}}$) from Scenario 1 to the SOCO data set and the derived best configurations for the SOCO set (C_{soco}). The tools are compared by their F-scores.

Tools	$C_{\text{generated}}$				C_{soco}		
	Settings	***T***	generated F-score	SOCO F-score	Settings	***T***	SOCO F-score
ccfx (C)	b=5,t=11	36	0.9760	0.8441	b={15 16 17}, t=12	25	0.9389
py-fuzzywuzzy	token_set_ratio	85	0.8757	0.6012	ratio	65	0.9338
jplag-java	t=7	19	0.8636	0.3168	t=12	29	0.9576
py-difflib	autojunk=false whitespace=false	28	0.8629	0.2113	autojunk=true whitespace=true	42	0.9443
simjava	r=16	15	0.8618	0.5888	r=25	23	0.9259
deckard (T)	M=30 S_1=2 S_2=0.95	17	0.8509	0.3305	M=50 S_1=1 S_2=1.0	19	0.9520
bzip2ncd	C=1..9	38	0.8494	0.3661	C=1 .. 9	54	0.8630
ncd-bzlib	N/A	37	0.8465	0.3357	N/A	52	0.8689
simian (C)	threshold=4, I_1	5	0.8413	0.6394	threshold=4, I_1	26	0.9593
ncd-zlib	N/A	30	0.8361	0.3454	N/A	57	0.8776

Note: M=mintoken, S_1=stride, S_2=similarity, I_1=ignoreVariableNames

Table 12: Results of using prec@n, ARP, and MAP over the generated data set with the tools' optimal configurations $C_{\text{generated}}$

Tool	Pair-based				Query-based			
	Set		Ranked		Ranked results			
	Rank	F-score	Rank	prec@n	Rank	ARP	Rank	MAP
ccfx	1	0.9760	1	0.9760	1	1.0000	1	1.0000
py-fuzzywuzzy	2	0.8757	3	0.8580	2	0.9150	2	0.9487
jplag-java	3	0.8636	5	0.8320	8	0.8510	8	0.9027
py-difflib	4	0.8629	10	0.8020	10	0.8380	9	0.8867
simjava	5	0.8618	2	0.8600	7	0.8580	6	0.9119
deckard	6	0.8509	7	0.8290	6	0.8740	7	0.9102
bzip2ncd	7	0.8494	5	0.8320	4	0.9090	4	0.9417
ncd-bzlib	8	0.8465	8	0.8260	3	0.9130	3	0.9434
simian	9	0.8413	4	0.8420	9	0.8480	10	0.8623
ncd-zlib	10	0.8361	9	0.8150	5	0.8980	5	0.9349

Table 13: Results of using prec@n, ARP, and MAP over the SOCO data set with the tools' optimal configurations C_{soco}

Tool	Pair-based				Query-based			
	Set		Ranked		Ranked results			
	Rank	F-score	Rank	prec@n	Rank	ARP	Rank	MAP
simian	1	0.9593	1	0.9536	1	0.9729	7	0.9693
jplag-java	2	0.9576	2	0.9514	2	0.9642	8	0.9659
deckard	3	0.9520	3	0.9470	3	0.9468	10	0.9473
py-difflib	4	0.9443	4	0.9360	4	0.9845	1	0.9903
ccfx	5	0.9389	7	0.8962	7	0.9642	9	0.9639
py-fuzzywuzzy	6	0.9338	5	0.9294	5	0.9826	4	0.9840
simjava	7	0.9259	6	0.9117	6	0.9859	2	0.9867
ncd-zlib	8	0.8776	8	0.8455	8	0.9853	3	0.9865
ncd-bzlib	9	0.8689	10	0.8322	10	0.9758	5	0.9813
bzip2ncd	10	0.863	9	0.8366	9	0.9770	6	0.9810

ranking of the ten tools changed slightly, as simjava (0.8600), simian (0.8410), and bzip2ncd (0.8310) perform better while py-difflib tool (0.8010) now performs much worse.

As illustrated in Figure 10, varying 15 n values from 100 to 1500, stepping up by 100, gave us an overview of how well the tools perform across different n sizes. The number of true positives is depicted by a dotted line. We could see that most of the tools performed really well in the very first few hundreds of top n results by having steady flat lines at prec@n of 1.0000. However, after passing the top 600 pairs, the performance of deckard, ncd-bzlib, ncd-zlib, simian and simjava started dropping. bzip2ncd, jplag-java, and py-fuzzywuzzy started reporting false positives after the top 700 pairs while py-difflib could stay until the top 800 pairs. ccfx was the only tool that could maintain 100% correct results until the top 900 pairs. After that, it also started reporting false positives. At the top 1,500 pairs, all the tools offered prec@n at approximately 0.6 to 0.7.

For the SOCO data set, we varied the n value from 100 to 800, also stepping up by 100,. The results in Table 13 used the n value of 453 which is the number of true positive in the corrected ground truth. We can clearly see that the ranking of 10 tools using prec@n closely resembles the ones using F-scores. Simian is the winner with the value 0.9536 followed by jplag-java (0.9514), deckard (0.9470), and py-difflib (0.9360). The ranking of these top four tools are exactly the same as using F-score. ccfx performs worse using prec@453 and moves down from 5th to 7th. Lastly, bzip2ncd (0.8366) performs slightly better and is ranked 9th using prec@453. ncd-bzlib (0.8322) obtains the lowest rank in the set. The overall performances of the tools across various n values is depicted in Figure 11 with the dotted line representing the number of true positives. The chart is somewhat analogous to the generated data set (Figure 10). Most of the tools started reporting false positives at the top 300 pairs except jplag-java, py-difflib, py-fuzzywuzzy and simjava. After the top 400 pairs, no tool could longer maintain 100% true positive results.

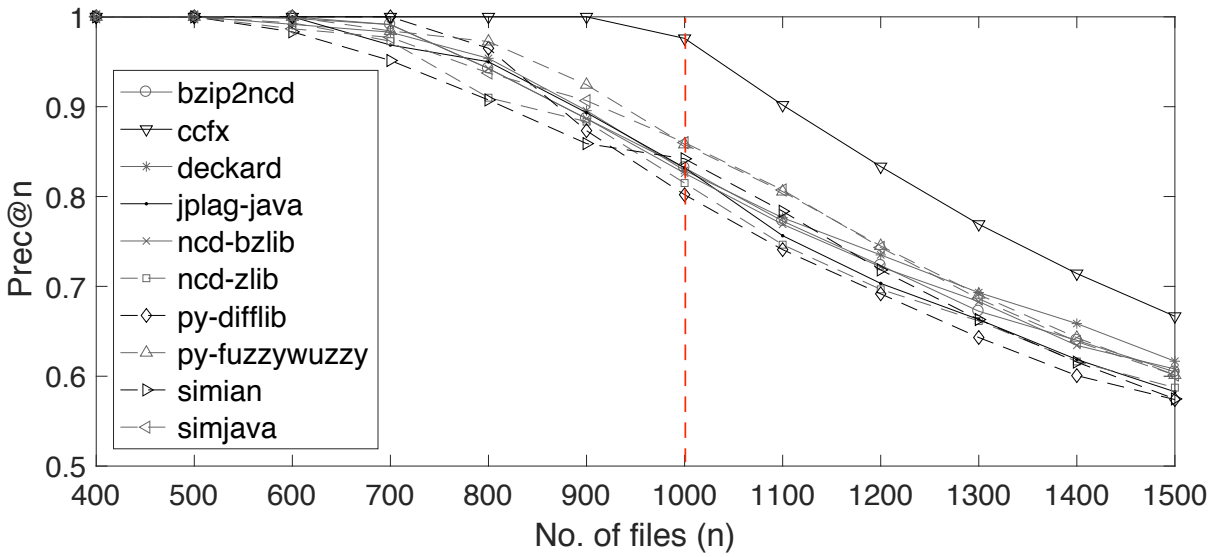

Fig. 10: Precision-at-n of the tools according to varied numbers of *n* against generated data set

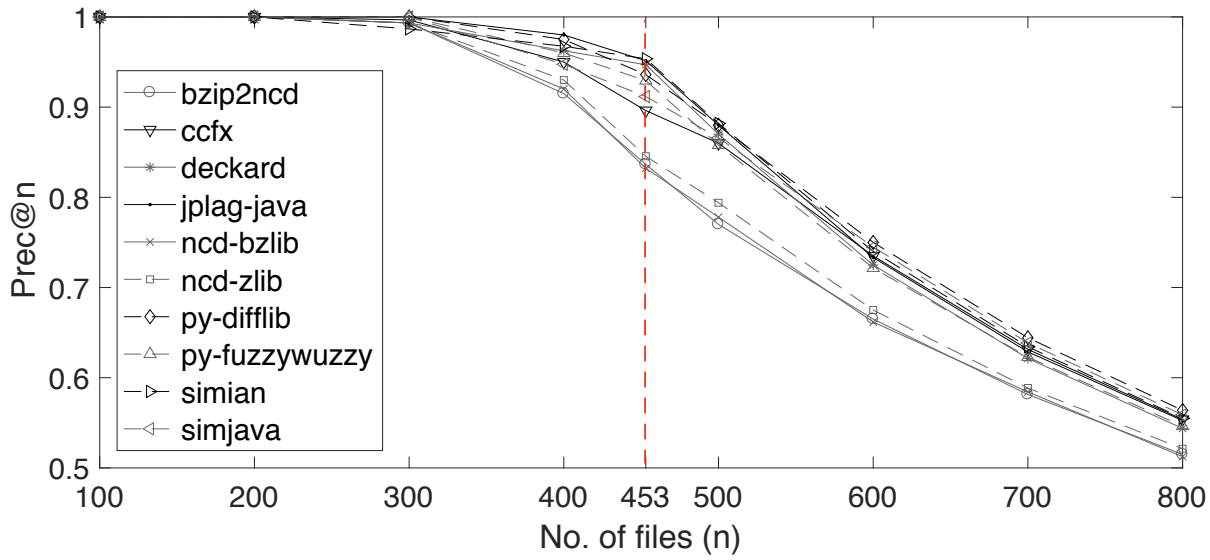

Fig. 11: Precision-at-n of the tools according to varied numbers of *n* against SOCO data set

Since prec@n is calculated from a set of top-*n* ranked results, its value shows how fast a tool can retrieve correct answers within a limited set of *n* most similar files. It also reflect how well the tool can differentiate between similar and dissimilar documents. A good tool would not be confused and produce a large gap of similarity value between true positive and true negative results. In this study, ccfx and simian have shown to be the best tools in terms of prec@n for pervasive modifications and boiler-plate code respectively.

Average *r*-Precision: ARP is a query-based error measure with presence of a ground truth. Since we already knew the ground truth of our two data sets, we did not need to vary the values of n as in prec@n. The n was set to the number of true positives.

For the generated data set, each file in the set of 100 files was used as a query once. Each query received 100 files ranked by their similarity values. We knew the ground truth that each file has 10 other similar files including itself (i.e. r or the number of relevant documents equals to 10). We cut off at the top 10 ranked results and calculated an r-precision value. Finally we computed ARP from an average of the 100 r-precisions. We reported the ARPs of the ten tools in Table 12. We can see that ccfx is still ranked the first with the perfect ARP of 1.0000 followed by py-fuzzywuzzy (0.9150). ncd-bzlib now performs much better using ARP (0.9130) and is ranked the third. The py-difflib (0.8380) is ranked last, similarly to its ranking by prec@n.

For SOCO, only files in the corrected ground truth were used as queries. This is because ARP can only be computed when relevant answers are retrieved. We found that the 453 pairs in the ground truth were formed by 115 unique files, and we used them as our queries. The number of r here was not fixed like in the generated data set. It depended on how many relevant answers exists in the ground truth for each particular query file and we calculated the r-precision based on that. The ARPs of the SOCO data set is reported in Table 13. Again, simian is ranked the first with ARP of 0.9729 followed by jplag-java (0.9642) and deckard (0.9468). ncd-bzlib, similar to the ranking by prec@n, stays at the last place.

ARP tells us how well the tools perform when we want all the true positive results in query-based manner. For example, one wants to find similar source code given an original source code that he or she possesses. They can use the original source code as a query and look for similar source files in a set of source code files that they have. In our study, ccfx is the best tool for this retrieval method against pervasive modifications and simian is the best tool for boiler-plate code.

Mean Average Precision: We decided to also include MAP in this study due to its well-known quality of discrimination and stability across several recall levels. It is also used when the ground truth of relevant documents is known. We computed MAP in a very similar way to ARP except that instead of only looking at the top r pairs, we calculated precision every time a new relevant source code file is retrieved. An average across all recall levels is then calculated. Lastly, the final average across all the queries is computed as MAP. We used the same number of relevant files to the ARP calculations for the generated and the SOCO data set. The results of MAP are reported in Table 12 and Table 13.

For the generated data set (Table 12), the rankings are very similar to ARP. ccfx (1.0000), py-fuzzywuzzy (0.9487), and ncd-bzlib (0.9434) are ranked the 1st, 2nd and 3rd while simian (0.8623) is the last one. For SOCO (Table 13), the rankings are very different from using F-score, prec@n, or ARP. The py-difflib (0.9903) becomes the best performing tool followed by simjava (0.9867) and ncd-zlib (0.9865). Deckard does not perform well on MAP and is ranked the last with the value of 0.9473.

MAP is similar to ARP because recall is taken into account. However, it differs from ARP by measuring precision at multiple recall levels. It is also different from F-score in terms of query-based measure instead of a pair-based measure. It shows how well a tool perform in average when it has to find all true positives for each query. In this study, the best peforming tool in terms of MAP is ccfx for pervasively modified code and py-difflib for boiler-plate code respectively.

5.4 Overall discussions

In summary, we have answered the 5 research questions after performing four experimental scenarios. We found that the state-of-the-art code similarity analysers perform differently on pervasively modified code. Properly configured, a well known and often used clone detector, ccfx, performed the best, closely followed by a Python string matching algorithm, fuzzywuzzy.

The experiment using compilation/decompilation for normalisation showed that compilation/decompilation is effective and greatly improves similarity detection techniques. Therefore, future implementations of clone or plagiarism detection tools or other similarity detection approaches could consider using compilation/decompilation for normalisation.

However, every technique and tool turned out to be extremely sensitive to its own configurations consisting of several parameter settings and a similarity threshold. Moreover, for some tools the optimal configurations turned out to be very different to the default configuration, showing one cannot just reuse (default) configurations.

Finding an optimal configuration is naturally biased by the particular data set. One cannot get an optimal results from their tools by directly applying the optimal derived parameter settings and similarity threshold from one data set to another data set. The SOCO data set, where we have applied the optimal configurations from the generated data set, clearly shows that configurations that claim to work well with a specific data set may not be guaranteed to work with future data sets. Researchers have to consider this limitation every time when they use similarity detection techniques in their studies.

We also compared the tools with the presence of boiler-plate code in the SOCO data set. We found that the simian clone detector performed the best followed by jplag-java and deckard clone detector. The optimal configurations derived from the SOCO data set were very different from the optimal configurations of pervasive modifications.

The chosen similarity threshold has the strongest impact on the results of similarity detection. We have investigated the use of three information retrieval error measures, precision-at-n, *r*-precision, and mean average precision, to remove the threshold completely and rely only on the ranked pairs. These three error measures are often used in information retrieval research but are rarely seen in code similarity measurement such as code clone or plagiarism detection. Using the three measures, we can see how successful the different techniques and tools are in distinguishing similar code from dissimilar code based on ranked results. The tool rankings can be used as guidelines to select tools in real-world scenarios of similar code search or code plagiarism detection. For example, one is interested in looking at only the top n most similar source code pairs due to limited time for manual inspection or one uses a file to query for the other most similar files.

5.5 Threats to validity

Internal validity: We carefully chose the data sets for our experiment. We created the first data set (generated) by ourselves to obtain the ground truth for positive and negative results. However, the obfuscators (Artifice and ProGuard) possibly may not represent typical pervasive modifications. The SOCO data set has been used in a competition for detecting reused code and a careful manual investigation has revealed errors in the provided ground truth that have been corrected.

Although we have attempted to use the tools with their best parameter settings, we cannot guarantee that we have and it may be possible that the poor performance of some detectors is due to wrong usage instead of the techniques used in the detector. Moreover, in this study we tried to compare the tools' performances based on several standard measurements of precision, recall, accuracy, F-score, AUC, prec@n, ARP and MAP. However, there might be some situations where other measurements are required and that might produce different results.

External validity: The tools used in this study are restricted to be open-source or at least be freely available, but they do cover several areas of similarity detection (including string-, token-, and tree-based approaches) and some of them are well-known similarity measurement techniques used in other areas such as normalised compression (information theory) and cosine similarity (information retrieval). Nevertheless, they might not be complete representatives of all available techniques and tools.

In addition, the two decompilers (Krakatau, Procyon) are only a subset of all decompilers available. So they may not totally represent the performance of the other decompilers in the market or even other source code normalisation techniques. However, we have chosen two instead of only one so we can compare their behaviours and performances. As we are exploiting features of Java source and byte code, our findings only apply to Java code.

6 Related Work

Plagiarism is obviously a problem of serious concern in education. Similarly in industry, the copying of code or programs is copyright infringement. They both affect the originality of one's idea, his or her credibility, and also the quality of their organisation. The problem of software plagiarism has been occurring for several decades in schools and universities (Cosma and Joy, 2008; Daniela et al, 2012) and in law, where one of the more visible cases regarding copyright infringement of software is the ongoing lawsuit between Oracle and Google (United States District Court, 2011).

To detect plagiarism or copyright infringement of source code, one has to measure similarity of two programs. Two programs can be similar at the level of purpose, algorithm, or implementation (Zhang et al, 2012). Most of software plagiarism tools and techniques focus on the level of implementation since it is most likely to be plagiarised. The process of code plagiarism involves pervasive modifications to hide the plagiarism which often includes obfuscation. The goal of code obfuscation is to make the modified code harder to understand by humans and harder to reverse engineer while preserving its semantics (Whale, 1990; Collberg et al, 2003, 2002, 1997). Deobfuscation attempts to reverse engineer obfuscated code (Udupa et al, 2005). Because Java byte code is comparatively high-level and easy to decompile, obfuscation of Java bytecode has focused on preventing decompilation (Batchelder and Hendren, 2007) while decompilers like Krakatoa (Proebsting and Watterson, 1997), Krakatau (Grosse, 2016) and Procyon (Strobel, 2016) attempt to decompile even in the presence of obfuscation.

Several similarity detection tools for source code and binary code have been introduced by the research community. Many of them are based on string comparison techniques such as Longest Common Subsequence (LCS) found in NICAD (Roy and Cordy, 2008), Plague (Whale, 1990), YAP (Wise, 1992), and CoP (Luo et al, 2014). Many tools transform source code into an intermediate representation such as tokens and apply similarity measurement on them (Plague (Whale, 1990), Sherlock (Joy et al, 2005), Sim (Gitchell and Tran, 1999), YAP3 (Wise, 1996), JPlag (Prechelt et al, 2002), CCFinder (Kamiya et al, 2002), CP-Miner (Li et al, 2006), iClones (Göde and Koschke, 2009), MOSS (Schleimer et al, 2003) and a few more (Burrows et al, 2007; Smith and Horwitz, 2009; Duric and Gasevic, 2012)). Structural similarity of cloned code can be discovered by using abstract syntax trees as found in CloneDR (Baxter et al, 1998) and Deckard (Jiang et al, 2007a) or by using program dependence graphs (Krinke, 2001; Komondoor and Horwitz, 2001; Liu et al, 2006). The transformation into an intermediate representation like a token stream or an abstract syntax tree can be seen as a kind of normalisation. NICAD (Roy and Cordy, 2008) uses pretty printing as part of the normalisation process for clone detection.

Although there is a large number of clone detectors, plagiarism detectors, and code similarity detectors invented in the research community, there are relatively few studies that compare and evaluate their performances. Bellon et al. (Bellon et al, 2007) proposed a framework for comparing and evaluating clone detectors and six tools (Dup, CloneDr, CCFinder, Duplix, CLAN, Duploc) were chosen for the studies. Later, Roy et al. (Roy et al, 2009) performed a thorough evaluation of clone detection tools and techniques covering a wider range of tools. However, they compare the tools and techniques using the evaluation results obtained from the tools' published papers without any real experimentation. Moreover, the performances in terms of recall for 11 modern clone detectors are evaluated based on four different code clone benchmark frameworks including Bellon's (Svajlenko and Roy, 2014). Hage et al. (Hage et al, 2010) compare five plagiarism detectors in term of their features and performances against 17 code modifications. Burd and Bailey (Burd and Bailey, 2002) compare five clone detectors for preventive maintenance task. Biegel et al. (Biegel et al, 2011) compare three code similarity measures to identify code that need refactoring.

Several code obfuscation methods can be found in (Luo et al, 2014). The techniques utilised include obfuscation by different compiler optimization levels, or using different compilers. Obfuscating tools exist at either source code level (e.g. Semantic Designs Inc.'s C obfuscator, Stunnix's CXX-obfuscator), and binary level (e.g. Diablo, Loco (Madou et al, 2006), CIL (Necula et al, 2002)).

An evaluation of code obfuscation techniques has been perform by Ceccato et al. (Ceccato et al, 2009). They evaluated how layout obfuscation by identifier renaming affects the participants' comprehension and ability to modify two given programs. They found that obfuscation by identifier renaming could slow down the attack by two to four times of the time needed by clear un-obfuscated programs. Their later study (Ceccato et al, 2013) still confirms that identifier renaming

is an effective obfuscation technique, even better than control-flow obfuscation by opaque predicates. Our two chosen obfuscators also perform layout obfuscation, including identifier renaming, in this study. However, instead of measuring understanding of obfuscated programs by human, we measure how well code similarity analysers perform on obfuscated code, which we use as a kind of pervasive code modifications. We also similarly decompiled obfuscated bytecodes and compared the tools' performances based on the decompiled source code.

There are studies that try to enhance performance of clone detectors by looking for more clones from the code's intermediate representation such as Jimple code (Selim et al, 2010), bytecode (Chen et al, 2014; Kononenko et al, 2014), or assembler code (Davis and Godfrey, 2010). Using intermediate representation for clone detection gives satisfying results by mainly increasing recall of the tools. Our study is different from them in the way that we apply decompilation as another code modification step before applying code similarity analysers. Our decompiled code is also Java source code and we can choose any source-based similarity analysers directly out of the box. Our results show that compilation/decompilation can also help improving tools' performances. An empirical study of using compilation/decompilation to enhance the performance of clone detection tool in three real-world system found similar results to our study (Ragkhitwetsagul and Krinke, 2017).

The work that is closest to ours is the empirical study of the efficiency of current detection tools against code obfuscation (Schulze and Meyer, 2013). The authors created the Artifice source code obfuscator and measured the effects of obfuscation on clone detectors. However, the number of tools chosen for the study was limited to only three detectors: JPlag, CloneDigger, and Scorpio. Nor has bytecode obfuscation been considered. The study showed that token-based clone detection outperformed text-, tree- and graph-based clone detection (similar to our findings).

Roy et al. (Roy and Cordy, 2009) use a mutation based approach to create a framework for the evaluation of clone detectors. However, their framework was mostly limited to locally confined modifications, only including systematic renaming as a pervasive modification. Due to the limitations, we haven't included their framework in our study. Moreover, they used their framework for a comparison limited to three variants of their own clone detector NICAD (Roy and Cordy, 2008).

Keivanloo et al. (Keivanloo et al, 2015) discussed the problem of using a single threshold for clone detection over several repositories and propose a solution using threshold-free clone detection based on unsupervised learning. The method mainly utilises k-means clustering with the Friedman quality optimization method. Our investigation of precision-at-n, ARP, and MAP focuses on the same problem but our goal is to compare the performance of several similarity detection tools instead of boosting the performance of one tool as in their study.

7 Conclusions

This study of comparing source code similarity analysers is the largest existing similarity detection study covering the widest range (30) of similarity detection techniques and tools to date. We found that the techniques and tools achieve varied performances when they are run against four different scenarios of modifications on source code. Our analysis provides a broad, thorough, performance-based evaluation of tools and techniques for similarity detection.

Our experimental results show that highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures for both pervasive code modifications and boiler-plate code. Moreover, we confirmed that compilation and decompilation can be used as an effective normalisation method that greatly improves similarity detection between Java source code, leading to three clone and plagiarism tools not reporting any false classification on our generated data set. Once again, our study showed that similarity detection techniques and tools are very sensitive to their parameter settings. One cannot just use default settings or re-use settings that have been optimised for one data set to another data set.

Importantly, the results of the study can be used as a guideline for researchers to select a proper technique with appropriate configurations for their data sets.

8 Acknowledgments

Chaiyong Ragkhitwetsagul is supported by the PhD scholarship from the Faculty of Information and Communication Technology, Mahidol University, Thailand.

References

Ahtiainen A, Surakka S, Rahikainen M (2006) Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In: Baltic Sea '06, pp 141–142

Batchelder M, Hendren L (2007) Obfuscating Java: The most pain for the least gain. In: Compiler Construction, pp 96–110

Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: ICSM'98, pp 368–377

Beitzel SM, Jensen EC, Frieder O (2009) Average R-Precision, Springer US, pp 195–195

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. Transactions on Software Engineering 33(9):577–591

Berghel HL, Sallach DL (1984) Measurements of program similarity in identical task environments. ACM SIGPLAN Notices 19(8):65

Biegel B, Soetens QD, Hornig W, Diehl S, Demeyer S (2011) Comparison of similarity metrics for refactoring detection. In: MSR '11

Breuer PT, Bowen JP (1994) Decompilation: the enumeration of types and grammars. Transactions on Programming Languages and Systems 16(5):1613–1647

Bruntink M, van Deursen A, van Engelen R, Tourwe T (2005) On the use of clone detection for identifying crosscutting concern code. Transactions on Software Engineering 31(10):804–818

Burd E, Bailey J (2002) Evaluating clone detection tools for use during preventative maintenance. In: SCAM '02, pp 36–43

Burrows S, Tahaghoghi SMM, Zobel J (2007) Efficient plagiarism detection for large code repositories. Software: Practice and Experience 37(2):151–175

Ceccato M, Di Penta M, Nagra J, Falcarin P, Ricca F, Torchiano M, Tonella P (2009) The effectiveness of source code obfuscation: An experimental assessment. In: ICPC '09, pp 178–187

Ceccato M, Di Penta M, Falcarin P, Ricca F, Torchiano M, Tonella P (2013) A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empirical Software Engineering pp 1040–1074

Chae DK, Ha J, Kim SW, Kang B, Im EG (2013) Software Plagiarism Detection: A Graph-based Approach. In: CIKM '13, pp 1577–1580

Chen K, Liu P, Zhang Y (2014) Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: ICSE '14, pp 175–186

Chow S, Chow S, Gu Y, Gu Y, Johnson H, Johnson H, Zakharov Va, Zakharov Va (2001) An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In: ISC '01, pp 144–155

Cifuentes C, Gough KJ (1995) Decompilation of binary programs. Software: Practice and Experience 25(7):811–829

Cilibrasi R, Vitányi PMB (2005) Clustering by compression. Transactions on Information Theory 51(4):1523–1545

Cilibrasi R, Cruz AL, de Rooij S, Keijzer M (2015) Complearn. http://complearn.org/index.html, accessed: 2016-02-14

Cohen A (2011) FuzzyWuzzy: Fuzzy string matching in Python. http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/, accessed: 2016-02-14

Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science University of Auckland

Collberg C, Myles G, Huntwork A (2003) Sandmark – a tool for software protection research. Security and Privacy 1(4):40–49

Collberg CS, Thomborson C, Member S (2002) Watermarking, tamper-proofing, and obfuscation – tools for software protection. Computer 28(8):735–746
Cosma G, Joy M (2008) Towards a definition of source-code plagiarism. Transactions on Education 51(2):195–200
Cosma G, Joy M (2012) An approach to source-code plagiarism detection and investigation using latent semantic analysis. Transactions on Computers 61(3):379–394
Crussell J, Gibler C, Chen H (2012) Attack of the Clones: Detecting Cloned Applications on Android Markets. In: ESORICS '12, pp 37–54
Crussell J, Gibler C, Chen H (2013) AnDarwin: Scalable Detection of Semantically Similar Android Applications. In: ESORICS '13, pp 182–199
Daniela C, Navrat P, Kovacova B, Humay P (2012) The issue of (software) plagiarism: A student view. Transactions on Education 55(1):22–28
Davey N, Barson P, Field S, Frank R, Tansley D (1995) The development of a software clone detector. International Journal of Applied Software Technology 1(3-4):219–36
Davies J, German DM, Godfrey MW, Hindle A (2013) Software bertillonage: Determining the provenance of software development artifacts. Empirical Software Engineering 18:1195–1237
Davis IJ, Godfrey MW (2010) From Where It Came: Detecting Source Code Clones by Analyzing Assembler. In: WCRE'10, pp 242–246
Desnos A, Gueguen G (2011) Android: From reversing to decompilation. Black Hat Abu Dhabi pp 1–24
Donaldson J, Lancaster A, Sposato P (1981) A plagiarism detection system. SIGCSE '81 pp 21–25
Ducasse S, Rieger M, Demeyer S (1999) A language independent approach for detecting duplicated code. In: ICSM '99, pp 109–118
Duric Z, Gasevic D (2012) A source code similarity system for plagiarism detection. The Computer Journal 56(1):70–86
Faidhi J, Robinson S (1987) An empirical approach for detecting program similarity and plagiarism within a university programming environment. Computers & Education 11(1):11–19
Flores E, Rosso P, Moreno L, Villatoro-Tello E (2014) Detection of source code re-use. http://users.dsic.upv.es/grupos/nle/soco/, accessed: 2016-02-14
Gibler C, Stevens R, Crussell J, Chen H, Zang H, Choi H (2013) AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In: MobiSys'13, p 431
Gitchell D, Tran N (1999) Sim: a utility for detecting similarity in computer programs. In: SIGCSE '99, pp 266–270
Göde N, Koschke R (2009) Incremental clone detection. In: CSMR'09, pp 219–228
Grier S (1981) A tool that detects plagiarism in Pascal programs. SIGCSE '81 13(1):15–20
Grosse R (2016) Krakatau bytecode tools. https://github.com/Storyyeller/Krakatau, accessed: 2016-02-14
GuardSquare (2015) ProGuard: bytecode obfuscation tool. http://proguard.sourceforge.net/, accessed: 2015-08-24
Hage J, Rademaker P, van Vugt N (2010) A comparison of plagiarism detection tools. Technical Report UU-CS-2010-015, Department of Information and Computing Sciences Utrecht University, Utrecht, The Netherlands
Halstead MH (1977) Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc.
Harris S (2015) Simian – similarity analyser, version 2.4. http://www.harukizaemon.com/simian/, accessed: 2016-02-14
Hartmann B, Macdougall D, Brandt J, Klemmer SR (2010) What would other programmers do? suggesting solutions to error messages. In: CHI '10, pp 1019–1028
Holmes R, Murphy GC (2005) Using structural context to recommend source code examples. In: ICSE '05, New York, New York, USA
Jiang L, Misherghi G, Su Z, Glondu S (2007a) DECKARD: Scalable and accurate tree-based detection of code clones. In: ICSE'07, pp 96–105
Jiang L, Su Z, Chiu E (2007b) Context-based detection of clone-related bugs. In: ESEC-FSE '07
Joy M, Luck M (1999) Plagiarism in programming assignments. Transactions on Education 42(2):129–133
Joy M, Griffiths N, Boyatt R (2005) The BOSS online submission and assessment system. Educational Resources in Computing 5(3)

Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: a multilinguistic token-based code clone detection system for large scale source code. Transactions on Software Engineering 28(7):654–670
Kapser C, Godfrey M (2006) "Cloning Considered Harmful" Considered Harmful. In: 2006 13th Working Conference on Reverse Engineering, pp 19–28
Kapser C, Godfrey MW (2003) Toward a taxonomy of clones in source code: A case study. In: ELISA '03, pp 67–78
Keivanloo I, Zhang F, Zou Y (2015) Threshold-free code clone detection for a large-scale heterogeneous java repository. In: SANER '15, pp 201–210
Komondoor R, Horwitz S (2001) Using slicing to identify duplication in source code. In: SAS'01, pp 40–56
Kononenko O, Zhang C, Godfrey MW (2014) Compiling Clones: What Happens? In: ICSME'14, pp 481–485
Krinke J (2001) Identifying similar code with program dependence graphs. In: WCRE '01, pp 301–309
Li M, Vitâanyi PMB (2008) An Introduction to Kolmogorov Complexity and Its Applications. Springer
Li Z, Lu S, Myagmar S, Zhou Y (2006) CP-Miner: Finding copy-paste and related bugs in large-scale software code. Transactions on Software Engineering 32(3):176–192
Lim Hi, Park H, Choi S, Han T (2009) A method for detecting the theft of Java programs through analysis of the control flow information. Information and Software Technology 51(9):1338–1350
Liu C, Chen C, Han J, Yu PS (2006) GPLAG: Detection of software plagiarism by program dependence graph analysis. In: KDD '06
Luo L, Ming J, Wu D, Liu P, Zhu S (2014) Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: FSE'14, pp 389–400
Madou M, Put LV, Bosschere KD (2006) Loco: An interactive code (de) obfuscation tool. In: PEPM'06, pp 140–144
Maebe J, Sutter BD (2006) Diablo. http://diablo.elis.ugent.be, accessed: 2016-02-14
Maletic J, Marcus A (2001) Supporting program comprehension using semantic and structural information. In: ICSE '01, pp 103–112
Manning CD, Raghavan P, Schutze H (2009) An Introduction toInformation Retrieval, vol 21. Cambridge University Press
McMillan C, Grechanik M, Poshyvanyk D (2012) Detecting similar software applications. In: ICSE'12, pp 364–374
Moreno L, Bavota G, Penta MD, Oliveto R, Marcus A (2015) How can i use this method? In: ICSE '15
Mycroft A (1999) Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). Programming Languages and Systems
Myles G, Collberg C (2004) Detecting Software Theft via Whole Program Path Birthmarks. In: ISC '04, pp 404–415
Nachenberg C (1996) Understanding and managing polymorphic viruses. The Symantec Enterprise Papers 30:16
Necula GC, Mcpeak S, Rahul SP, Weimer W (2002) CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC'02, pp 213–228
Ottenstein K (1976) An algorithmic approach to the detection and prevention of plagiarism. SIGCSE '76 8(4):41
Pate JR, Tairas R, Kraft NA (2013) Clone evolution: A systematic review. Journal of software: Evolution and Process 25:261–283
Pavlov I (2016) 7-Zip. http://www.7-zip.org, accessed: 2016-02-14
Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al (2011) Scikit-learn: Machine learning in python. Journal of Machine Learning Research 12(Oct):2825–2830
Pike R, Loki (2002) The Sherlock plagiarism detector. http://www.cs.usyd.edu.au/~scilect/sherlock/, accessed: 2016-02-14
Poulter G (2012) Python ngram 3.3. https://pythonhosted.org/ngram/, accessed: 2016-02-14
Prechelt L, Malpohl G, Philippsen M (2002) Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science 8(11):1016–1038
Proebsting Ta, Watterson Sa (1997) Krakatoa: Decompilation in Java (does bytecode reveal source?). In: USENIX, pp 185–198
Python Software Foundation (2016) difflib – helpers for computing deltas. http://docs.python.org/2/library/difflib.html, accessed: 2016-02-14

Ragkhitwetsagul C, Krinke J (2017) Using Compilation / Decompilation to Enhance Clone Detection. In: 11th International Workshop on Software Clone

Ragkhitwetsagul C, Krinke J, Clark D (2016) Similarity of Source Code in the Presence of Pervasive Modifications. In: SCAM '16

Roy CK, Cordy JR (2008) NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: ICPC'08, pp 172–181

Roy CK, Cordy JR (2009) A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In: ICSTW '09, pp 157–166

Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming 74(7):470–495

Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV (2016) SourcererCC: Scaling Code Clone Detection to Big-Code. In: ICSE '16, pp 1157–1168

Schleimer S, Wilkerson DS, Aiken A (2003) Winnowing: Local algorithms for document fingerprinting. In: SIGMOD'03

Schulze S, Meyer D (2013) On the robustness of clone detection to code obfuscation. In: IWSC'13, pp 62–68

Selim GM, Foo KC, Zou Y (2010) Enhancing Source-Based Clone Detection Using Intermediate Representation. In: WCRE'10, pp 227–236

Semantic Designs (2016) Thicket™ family of source code obfuscators. http://www.semdesigns.com/Products/Obfuscators/, accessed: 2016-02-14

Smith R, Horwitz S (2009) Detecting and measuring similarity in code clones. In: IWSC'09

Strobel M (2016) Procyon / Java Decompiler. https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler, accessed: 2016-02-14

Stunnix (2016) http://stunnix.com, accessed: 2016-02-14

Svajlenko J, Roy CK (2014) Evaluating modern clone detection tools. In: ICSME '14, pp 321–330

Tamada H, Okamoto K, Nakamura M (2004) Dynamic software birthmarks to detect the theft of windows applications. In: ISFST '04

Tian Z, Zheng Q, Liu T, Fan M, Zhang X, Yang Z (2014) Plagiarism detection for multithreaded software based on thread-aware software birthmarks. In: ICPC '14, pp 304–313

Turk J, Stephens M (2016) A python library for doing approximate and phonetic matching of strings. https://github.com/jamesturk/jellyfish, accessed: 2016-02-14

Udupa SK, Debray SK, Madou M (2005) Deobfuscation: reverse engineering obfuscated code. In: WCRE '05, pp 45–56

United States District Court (2011) Oracle America, Inc. v. Google Inc., No. 3:2010cv03561 – Document 642 (N.D. Cal. 2011). http://law.justia.com/cases/federal/district-courts/california/candce/3:2010cv03561/231846/642/, accessed: 2016-02-14

Wang CWC, Davidson J, Hill J, Knight J (2001) Protection of software-based survivability mechanisms. In: DSN '01

Wang T, Harman M, Jia Y, Krinke J (2013) Searching for better configurations: A rigorous approach to clone evaluation. In: FSE'13, pp 455–465

Whale G (1990) Identification of program similarity in large populations. The Computer Journal 33(2):140–146

Wise MJ (1992) Detection of similarities in student programs. In: SIGCSE '92, pp 268–271

Wise MJ (1996) YAP3: Improved detection of similarities in computer program and other texts. In: SIGCSE '96, pp 130–134

Zhang F, Jhi YC, Wu D, Liu P, Zhu S (2012) A first step towards algorithm plagiarism detection. In: ISSTA '12

Zhang F, Huang H, Zhu S, Wu D, Liu P (2014) ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In: WiSec '14, pp 25–36