CrossMark

A comparison of code similarity analysers

Chaiyong Ragkhitwetsagul[1] (ORCID) · **Jens Krinke[1]** ·
David Clark[1]

Abstract Copying and pasting of source code is a common activity in software engineering. Often, the code is not copied as it is and it may be modified for various purposes; e.g. refactoring, bug fixing, or even software plagiarism. These code modifications could affect the performance of code similarity analysers including code clone and plagiarism detectors to some certain degree. We are interested in two types of code modification in this study: pervasive modifications, i.e. transformations that may have a global effect, and local modifications, i.e. code changes that are contained in a single method or code block. We evaluate 30 code similarity detection techniques and tools using five experimental scenarios for Java source code. These are (1) pervasively modified code, created with tools for source code and bytecode obfuscation, and boiler-plate code, (2) source code normalisation through compilation and decompilation using different decompilers, (3) reuse of optimal configurations over different data sets, (4) tool evaluation using ranked-based measures, and (5) local + global code modifications. Our experimental results show that in the presence of pervasive modifications, some of the general textual similarity measures can offer similar performance to specialised code similarity tools, whilst in the presence of boiler-plate code, highly specialised source code similarity detection techniques and tools outperform 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. Moreover, we demonstrate that optimal configurations are very sensitive to a specific data

---

Communicated by: Michaela Greiler and Gabriele Bavota

✉ Chaiyong Ragkhitwetsagul
chaiyong.ragkhitwetsagul.14@ucl.ac.uk

Jens Krinke
j.krinke@ucl.ac.uk

David Clark
david.clark@ucl.ac.uk

[1] Centre for Research on Evolution, Search and Testing, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK

set. After directly applying optimal configurations derived from one data set to another, the tools perform poorly on the new data set. The code similarity analysers are thoroughly evaluated not only based on several well-known pair-based and query-based error measures but also on each specific type of pervasive code modification. 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. Whilst 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).

1.1 Motivation

The assessment of source code similarity has a co-evolutionary relationship with the modifications made to the code at the point of its creation. 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. (2007) proposed a framework for comparing and evaluating 6 clone detectors, Roy et al. (2009) evaluated a large set of clone detection tools but only based on results obtained from the tools' published papers, Hage et al. (2010) compare five plagiarism detectors against 17 code modifications, Burd and Bailey (2002) compare five clone detectors for preventive maintenance tasks, Biegel et al. (2011) compare three code similarity measures to identify code that needs refactoring, Svajlenko and Roy (2016) developed and used a clone evaluation framework called BigCloneEval to evaluate 10 state-of-the-art clone detectors. Although these studies cover various goals of tool evaluation and cover the different types of code modification found in the chosen data sets, they suffer from two limitations: (1) the selected tools are limited to only a subset of clone or plagiarism detectors, and (2) the results are based on different data sets, so one cannot compare a tool's performance from one study to another tool's from another study. To the best of our knowledge, there is no study that performs a comprehensive and fair comparison of widely-used code similarity analysers based on the same data sets.

In this paper, we fill the gap by presenting the largest extant study on source code similarity that covers the widest range of techniques and tools. We study the tools' performances on both local and pervasive (global) code modifications usually found in software engineering activities such as code cloning, software plagiarism, and code refactoring. This study is motivated by the question:

"When source code is copied and modified, which code 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 using several error measures. The aim of this study 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 on source code with local and global modifications. 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 clone detectors' parameters using search-based techniques. Therefore, whilst including more tools in this study, we have also searched through a wider range of configurations for each tool, studied their impact, and discovered the best configurations for each data set in our experiments. After obtaining tools' optimal configurations derived from one data set, we apply them to another data set and observe if they can be reused effectively.

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.

1.2 Contributions

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 five 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, in overall, highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures. However, we also observed some situations where compression-based, and textual similarity tools are recommended over clone and plagiarism detectors.

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 improve the effectiveness of similarity measurement techniques with statistical significance. Three of the similarity detection techniques and tools reported no false classifications once such normalisation was applied.

3. Data set of pervasive code modifications: The generated data set with pervasive modifications used in this study has been created to be challenging for code similarity analysers. According to the way we constructed the data set, the complete ground truth is known. We make the data set publicly available so that it can be used in future studies of tool evaluation and comparison.

Compared to our previous work (Ragkhitwetsagul et al. 2016), we have expanded the study further as follows. First, we doubled the size of the 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, and a data set of boiler-plate code with pervasive modifications. Since boiler-plate code is inherently different from pervasively modified code and normally found in software development (Kapser 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 Source Code Modifications

We are interested in two scenarios of code modifications in this study: pervasive code modifications (global) and boiler-plate code (local). Their definitions are as follows.

Pervasive modifications are code changes that affect the code globally across the whole file with multiple changes applied one after another. These are code transformations that are mainly found in the course of software plagiarism when one wants to conceal copied code by changing their appearance and avoid detection (Daniela et al. 2012). Nevertheless, they also represent code clones that are repeatedly modified over time during software evolution (Pate et al. 2013), and source code before and after refactoring activities (Fowler 2013). However, our definition of pervasive modifications excludes strong obfuscation (Collberg et al. 1997), that aims to protect code from reverse engineering by making it difficult or impossible to understand.

Most clone or plagiarism detection tools and techniques tolerate different degrees of change and still identify cloned or plagiarised fragments. However, whilst they usually have no problem in the presence of local or confined modifications, pervasive modifications that transform whole files remain a challenge (Roy and Cordy 2009). For example, in a situation that multiple methods are merged into a single method due to a code refactoring activity. A clone detector focusing on method-level clones would not report the code before and after merging as a clone pair. Moreover, with multiple lexical and structural code changes applied repeatedly at the same time, resulting source code can be totally different. When one looks at code before and after applying pervasive modifications, one might not be able to tell that both originate from the same file. We found that code similarity tools have the same confusion.

We define source code with pervasive modifications to contain a combination of the following code changes:

1. Lexical changes of formatting, layout modifications (Type I clones), and identifier renaming (Type II clones).
2. Structural changes, e.g. `if` to `case` or `whilst` to `for`, or insertions or deletions of statements (Type III clones).

```
/* original */
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo, j = hi+1;
    Comparable v = a[lo];
    while (true)
    {
        while (less(a[++i], v)) if (i == hi) break;
        while (less(v, a[--j])) if (j == lo) break;
        if (i >= j) break;
            exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}

/* plagiarised code */
private static int partition(int[] bob, int left, int right) {
    int x = left;
    int y = right+1;
    for (;;) {
        while (less(bob[left], bob[--y]))
            if (y == left) break;
        while (less(bob[++x], bob[left]))
            if (x == right) break;
        if (x >= y) break;
            swap(bob, y, x);
    }
    swap(bob, y, left);
    return y;
}
```

Fig. 1 Pervasive modifications found in a programming submission

3. Extreme code transformations that preserve source code semantics but change its syntax
    (Type IV clones).

Figure 1 shows an example of code before and after applying pervasive modifications. It
is a real-world example of plagiarism from a university's programming class submission.[1]
Boiler-plate code occurs when developers reuse a code template, usually a function or a
code block, to achieve a particular task. It has been defined as one of the code cloning pat-
terns by (Kapser 2006; Kapser and Godfrey 2008). Boiler-plate code can be found when
building device drivers for operating systems (Baxter et al. 1998), developing android appli-
cations (Crussell et al. 2013), and giving programming assignments (Burrows et al. 2007;
Schleimer et al. 2003). Boiler-plate code usually contains small code modifications in order
to adapt the boiler-plate code to a new environment. In contrast to pervasive modifications,
the modifications made to boiler-plate code are usually contained in a function or block.
Figure 2 depicts an example of boiler-plate code used for creating new HTTP connection
threads which can be reused as-is or with minimum changes.

2.2 Code Similarity Measurement

Since the 1970s, myriads of tools have been introduced to measure the similarity of source
code. They are used to tackle problems such as code clone detection, software licencing

---

[1]https://www.princeton.edu/pr/pub/integrity/pages/plagiarism/

```
private void createConnectionThread(int input)
{
  data = new HoldSharedData(startTime, password, pwdCounter);

  int numOfThreads = input;
  int batch = pwdCounter/numOfThreads + 1;
  numOfThreads = pwdCounter/batch + 1;
  System.out.println("Number_of_Connection_Threads_Used=" + numOfThreads);
  ConnectionThread[] connThread = new ConnectionThread[numOfThreads];

  for(int index = 0; index < numOfThreads; index ++)
  {
    connThread[index] = new ConnectionThread(url, index, batch, data);
    connThread[index].conn();
  }
}
```

Fig. 2 A boiler-plate code to create connection threads

violation, and software plagiarism. The tools utilise different approaches to computing 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 found empirically 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, whilst 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 methods 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) (Dangel A and Pelisse R 2011).

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 abstraction level can be adjusted by defining the types of tokens. Depending on how the tokens are defined, the token stream may normalise textual differences and capture 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. (2007), and the Source Code Similarity Detector System (SCSDS) (Duric and Gasevic 2013). The token-based representation is widely used in source code similarity measurement and very efficient on 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 a widely-used structure when computing program similarity by finding similar subtrees between two ASTs. The capability of comparing programs' structures allows tree-based tools to locate similar code with a 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 (2001), Komondoor and Horwitz (2001), Chae et al. (2013) and Chen et al. (2014). Although the tools demonstrate 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 unavailable. 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. 2012, 2013; Tian et al. 2014; Tamada et al. 2004; Myles and Collberg 2004; Hi et al. 2009; Zhang et al. 2012, 2014; McMillan et al. 2012; Luo et al. 2014).

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 further possibilities in 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).

2.3 Obfuscation and Deobfuscation

Obfuscation is a mechanism of making changes to a program whilst preserving its original functions. It originally aimed 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. 1997, 2002). Lexical transformations can be achieved by renaming identifiers and formatting changes, whilst 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 make the source code difficult to reverse engineer. Similarly, binary-code obfuscators transform the content of executable files.

Many obfuscation techniques have been invented and put to use in commercial obfuscators. 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. (2001) propose a sophisticated deep obfuscation method called *control flow flattening* which is 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), replacing 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 malware (Nachenberg 1996) or as a resiliency test for a newly developed obfuscation method (Madou et al. 2006). Whilst *surface obfuscation* such as variable renaming can be handled straightforwardly, *deep obfuscation* which 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) .

2.4 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. (2014). They found that 13.51% of all applications from five different Android markets are clones. Gibler et al. (2013) discovered that these decompiled and cloned apps can divert advertisement 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 2017b) 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 the decompilation step.

3 Empirical Study

Our empirical study consists of five experimental scenarios covering different aspects and characteristics of source code similarity. Three experimental scenarios examined tool/technique performance on three different data sets to discover any strengths and weaknesses. These three are (1) experiments on the products of the two obfuscation tools, (2) experiments on an available data set for identification of reuse boiler-plate code (Flores et al. 2014), and (3) experiments on the combinations of pervasive modifications and boiler-plate code. The fourth scenario examined the effectiveness of compilation/decompilation as a preprocessing normalisation strategy and the fifth 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 and boiler-plate code?* We compare 30 code similarity analysers using a data set of 100 pervasively modified pieces of source code and a data set of 259 pieces of Java source code that incorporate reused boiler-plate 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 so that we can fairly compare the techniques. We are also interested to see if one can gain optimal performance of the tools by relying on default configurations.

RQ3 (Normalisation by decompilation): *How much does compilation followed by decompilation as a pre-processing normalisation method improve detection results for pervasively modified code?* 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 effectively reuse optimal tool configurations for one data set on another data set?* We apply the optimal tool configurations obtained using one data set when using the tools with another data set and investigate whether they still offer the tools' best performances.

RQ5 (Ranked Results): *Which tools perform best when only the top n results are retrieved?* Besides the set-based error measures normally used in clone and plagiarism detection evaluation (e.g. precision, recall, F-scores), we also compare and report the tools'

performances using ranked results adopted from information retrieval. This comparison has a practical benefit in terms of plagiarism detection, manual clone study, and automated software repair.

RQ6 (Local + global code modifications): *How well do the techniques perform when source code containing boiler-plate code clones have been pervasively modified?* We evaluate the tools on a data set combining both local and global code modifications. This question also studies which types of pervasive modifications (source code obfuscation, bytecode obfuscation, compilation/decompilation) strongly affect tools' performances.

3.1 Experimental Framework

The general framework of our study, as shown in Fig. 3, 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 $\text{sim}(x, y)$ 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 $\text{Sim}(F)$ out of all files F is

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

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 whilst it is actually dissimilar (false positive, *FP*), and incorrectly classified as dissimilar pair whilst 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.

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
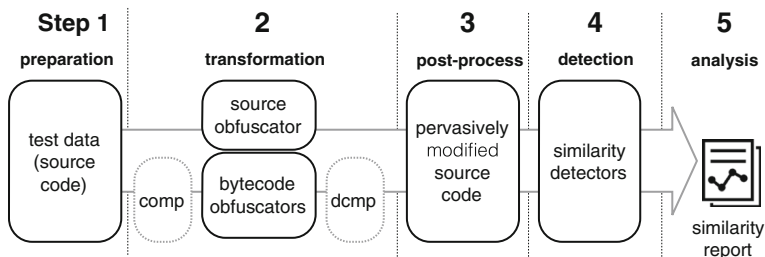

Fig. 3 The experimental framework

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 `whilst` 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, merging methods/classes and modifying package access permissions.

Using source and bytecode obfuscators, we can create pervasively modified source code that contains modifications of lexical and structural changes. We have investigated the code transformations offered by Artifice and ProGuard and found that they cover changes commonly found in both code cloning and code plagiarism as reported by Roy and Cordy (2009), Schulze and Meyer (2013), Duric and Gasevic (2013), Joy and Luck (1999), and Brixtel et al. (2010). The details of code modifications supported by our obfuscators are shown in Table 1.

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

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.

Table 1 List of pervasive code modifications offered by our source code and bytecode obfuscator, and compiler/decompilers

Code modifications	Artifice	ProGuard	(De)compilers
Lexical modification			
Formatting changes (Roy and Cordy 2009; Duric and Gasevic 2013; Joy and Luck 1999)	✓		✓
Addition, modification or deletion of comments (Duric and Gasevic 2013; Joy and Luck 1999)	✓		✓
Renaming of identifiers, methods (Roy and Cordy 2009; Duric and Gasevic 2013; Joy and Luck 1999; Brixtel et al. 2010; Fowler 2013)	✓	✓	✓
Modification of constant values (Duric and Gasevic 2013)		✓	
Structural modification			
Split or merge of variable declarations (Duric and Gasevic 2013)	✓		✓
Addition, modification or deletion of modifiers (Duric and Gasevic 2013; Fowler 2013)		✓	✓
Line insertion/deletion with further edits (Roy and Cordy 2009)	✓		✓
Reordering of statements & control replacements (Roy and Cordy 2009; Duric and Gasevic 2013; Joy and Luck 1999; Brixtel et al. 2010)	✓	✓	✓
Modification of control structures (Duric and Gasevic 2013; Joy and Luck 1999; Brixtel et al. 2010)	✓		✓
Changing of data types and modification of data structures (Duric and Gasevic 2013)			✓
Method inlining and method refactoring (Duric and Gasevic 2013; Fowler 2013)		✓	
Structural redesign of source code (Duric and Gasevic 2013; Fowler 2013)			✓

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 class file 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.

Using a combination of compilation and decompilation to generate code with pervasive modifications can represent source code that has been refactored (Fowler 2013), or rewritten (i.e. Type IV clones) (Roy et al. 2009). Whilst its semantics has been preserved, the source code syntax including layout, variable names, and structure may be different. Table 1 shows code modifications that are supported by our compiler and decompilers.

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 comes in versions for text (jplag-text and simtext) and Java (jplag-java and simjava), whilst Sherlock (Pike R 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, $sim(x, y)$, 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 between x and y found in x (overlaps are handled), i.e. $frag_i^x(x, y)$, to the size of x and vice versa.

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

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 the clone reports only contain clone lines, the actual implementation of clone detection tools works at a different granularity of code fragments. Measuring clone similarity at a single granularity level, such as line, may penalise some tools whilst favouring 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 whilst CCFinderX and iClones work at token level. However, Deckard compares clones based on ASTs so its similarity comes 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. Using these three levels 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-grammes, FuzzyWuzzy (Cohen 2011) uses fuzzy string token 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

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

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

The result of $\mathrm{sim}_D(x, y)$ is asymmetric as it depends on the size of the denominator. Hence $\mathrm{sim}_D(x, y)$ usually produces a different result from $\mathrm{sim}_D(y, x)$. This is because $\mathrm{sim}_D(x, y)$ 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 2. The default configurations of each tools, as displayed in Table 3, 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 3.

4 Experimental Scenarios

To answer the research questions, five experimental scenarios have been designed and studied following the framework presented in Fig. 3. 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 24 Microsoft Azure virtual machines with up to 16 cores, 56 GB memory 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, finally, `SqrtAlgorithm`. We downloaded them from two programming websites as shown in Table 4 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 usually discussed in basic programming classes. The process of test

Table 2 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 R 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-gramme
cosine (Pedregosa et al. 2011)	cosine similarity from machine learning library

data preparation and transformation is illustrated in Fig. 5. 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 Fig. 4 (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 Fig. 5 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

Table 3 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
iclones	minblock	min token length	20	8 10 20 30 40 50
	minclone	min no. of tokens	100	50 60 .. 140 150
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

Table 3 (continued)

Tool	Settings	Details	DF	Range
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			
difflib	autojunk	auto. junk heuristic	N/A	true, false
	whitespace	ignoring white space	N/A	true, false
fuzzywuzzy	similarity	similarity calculation	N/A	ratio, partial_ratio, token_sort_ratio, token_set_ratio
jellyfish	distance	edit distance algo.	N/A	jaro_distance, jaro_winkler
ngram	N/A			
cosine	N/A			

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 Fig. 3, 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 3. Hence, the number

Table 4 Descriptions of the 10 original Java classes in the generated data set

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

[a]classes downloaded from http://www.softwareandfinance.com/Java

[b]classes downloaded from http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures

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

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

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

Fig. 4 The same code fragments, a constructor of `MagicSquare`, after pervasive modifications, and compilation/decompilation

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 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 (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.
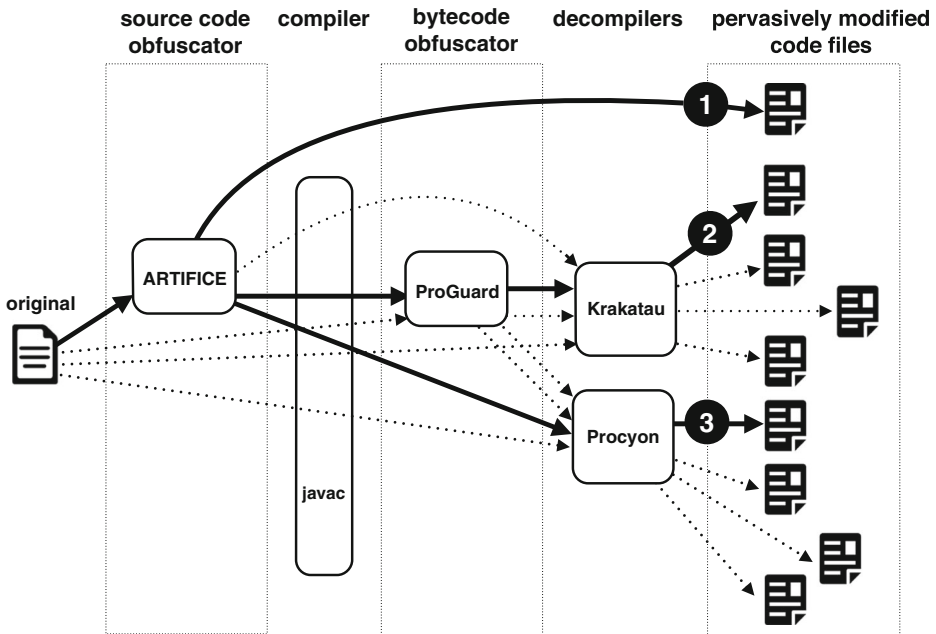
Fig. 5 Test data generation process

As shown in Table 5, 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' 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.[2]

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):

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

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

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 Fig. 6, 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".

---

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

Table 5 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^{decomp}). The SOCO data set is used in Scenario 3 and the SOCO with pervasive modification (SOCO^{gen}) is used in Scenario 5

Scenario	Data set	Files	#Comparisons	Positives	Negatives
1	generated	100	10,000	1,000	9,000
2	generated^{decomp}	100	10,000	1,000	9,000
3	SOCO	259	67,081	453	66,628
3	SOCO^{gen}	330	20,691	1,045	19,646

4.2 Scenario 2 (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 Fig. 3). 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 answer key of true clone pairs is provided. The answer key 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 answer key contained two large clusters in which
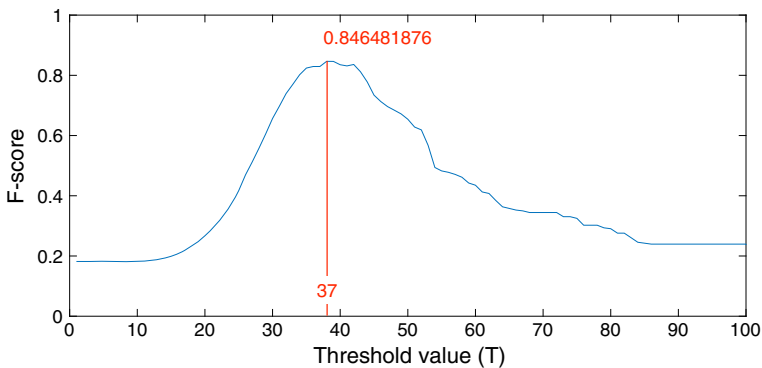

Fig. 6 The graph shows the F-score and the threshold values of ncd-bzlib. The tool reaches the highest F-score when the threshold equals 37

pairs were missing and that two given pairs were wrong.[3] After removing the wrong pairs and adding the missing pairs, the corrected ground truth contains $259 + 2 \times 97 = 453$ 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.

Since the SOCO data set is 2.59 times larger than the generated data set (259 Java files vs. 100 Java files), it takes much longer to run. For example, it took CCFinderX 7 hours 48 minutes[4] to complete $259^2 = 67,081$ pairwise comparisons with one of its configurations on our Azure virtual machine. To complete the search space of $20 \times 14 = 280$ CCFinderX's configurations, it took us 90 days. Executions of the 30 tools with all of their possible configurations cover 99,816,528 pairwise comparisons in total for this data set compared to 14,880,000 comparisons in Scenario 1. We analysed 1,448 similarity reports in total.

4.3 Scenario 3 (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 `whilst` loops are replaced by the same `if` and `goto` structures at the bytecode level.

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 $\text{sim}(P', Q')$ and get more accurate results than $\text{sim}(P, Q)$.

In this scenario, we focus on the generated data set containing pervasive code modifications 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.

4.4 Scenario 4 (Ranked Results)

In our three previous scenarios, we compared the tools' performances using their optimal F-scores. The F-score offers a 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

---

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

[4]User time measured by `/usr/bin/time -p` command.

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. Moreover, this approach also has applications in software engineering including finding candidates for plagiarism detection, automated software repair, working code examples, and large-scale code clone detection.

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 ground truth, we can set the value of n to be the number of relevant results (i.e. true positives). With a known 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) measures the quality of results across several recall levels where each relevant result is returned. 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:

$$\text{aprec@}n_q = \frac{1}{|R_q|} \sum_{i=1}^{|R_q|} \text{prec@}n_{q_i}$$

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 generating similarity pairs, and 2) ranking the results based on their similarities reported by the tools. The higher the

similarity value, the higher the rank. The top ranked result has the highest similarity value. If a tie happens, we resort to a 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 uses this method to report plagiarised source code pairs (Prechelt et al. 2002). Moreover, finding the most similar files is useful in a manual study of large-scale code clones (e.g. in a study by Yang et al. (2017)) when too many clones are reported and researchers are only feasibly able to investigate by hand a few of the most similar clone candidates.

ARP and MAP are calculated by considering the question *"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 a 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 results to only the top 10, since our ground truth contained 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. The query-based approach is suitable when one does not require the retrieval of all the similar pairs of code but only the most relevant ones for a given query. This situation occurs when performing code search for automated software repair (Ke et al. 2015). One may not feasibly try all returned repair candidates but only the top-ranked ones. Another example is searching for working code examples (Keivanloo et al. 2014) when one wants to pick only the top ranked solution.

Using these three error measures, we can compare performances of the similarity detection techniques and tools without relying on the threshold at all. They also provide another aspect of evaluating the tools' performances by observing how well the tools report correct results within the top *n* pairs.

4.5 Scenario 5 (Pervasive Modifications + Boiler-Plate Code)

We have two objectives for this experimental scenario. First, we are interested in a situation where local and global code modifications are combined together. This is done by applying pervasive modifications on top of reused boiler-plate code. This scenario occurs in software plagiarism when only a small fragment of code is copied and later pervasive modifications are applied to the whole source code to conceal the copied part of the code. It also represents a situation where a boiler-plate code has been reused and repeatedly modified (or refactored) during software evolution. We are interested to see if the tools can still locate the reused boiler-plate code. Second, we shift our focus from measuring how well our tools find *all* similar pairs of pervasively modified code pieces, as we did in Scenario 1, to measuring how well our tools find similar pairs of code pieces based on *each* pervasive code modification type. This is a finer-grained result and provides insights into the effects of each pervasive code modification type on code similarity. The default configurations are chosen for this experimental scenario to reflect a real use case when one does not know the optimal configurations of the tools and also to show the effect of each pervasive code modifications

on the tools' performances when they are picked off-the-shelf without any tuning. Since some threshold needs to be chosen, we used the optimal threshold for each tool.

We use the data set called SOCO^{gen} which is derived from the SOCO data set used in Scenario 3. We follow the 5 steps in our experimental framework (see Fig. 3) by using the SOCO's data set with boiler-plate code as a test data (Step 1). Amongst 259 SOCO files, 33 are successfully compiled and decompiled after code obfuscations by our framework. Each of the 33 files generates 10 pervasively modified files (including itself) resulting in 330 files available for detection (Step 4). The statistics of SOCO^{gen} is shown in Table 5.

We change the similarity detection in Step 4 to focus only on comparing modified code to their original. Given M as a set of the 10 pervasive code modification types, a set of similar pairs of files $\text{Sim}_m(F)$ out of all files F with a pervasive code modification m is

$$
\begin{aligned}
M &= \{O, A, K, P_c, P_g K, P_g P_c, AK, AP_c, AP_g K, AP_g P_c\} \\
\text{Sim}_m(F) &= \{(x, y) \in F_O \times F_m : m \in M; \text{sim}(x, y) > T\}
\end{aligned}
\tag{6}
$$

Table 6 presents the 10 pervasive code modification types; including the original (O), source code obfuscation by Artifice (A), decompilation by Krakatau (K), decompilation by Procyon (P_c), bytecode obfuscation by ProGuard and decompilation by Krakatau ($P_g K$), bytecode obfuscation by ProGuard and decompilation by Procyon ($P_g P_c$), and four other combinations (AK, AP_c, $AP_g K$, $AP_g P_c$); and ground truth for each of them. The number of code pairs and true positive pairs of A to $AP_g P_c$ are twice larger than the Original (O) type because of asymmetric similarity between pairs, i.e. $\text{Sim}(x, y)$ and $\text{Sim}(y, x)$.

We measured the tools' performance on each $\text{Sim}_m(F)$ set. By applying tools on a pair of original and pervasively modified code, we measure the tools based on one particular type of code modifications at a time. In total, we made 620,730 pairwise comparisons and analysed 330 similarity reports in this scenario.

5 Results

We used the five experimental scenarios of pervasive modifications, decompilation, reused boiler-plate code, ranked results, and the combination of local and global code modification to answer the six research questions. The execution of 30 similarity analysers on the data

Table 6 10 pervasive code modification types

| Type | Modification | Obfuscation | | Decomp. | Pairs | TP |
		Source	Bytecode			
O	Original				1,089	55
A	Artifice	✓			2,178	110
K	Krakatau			✓	2,178	110
P_c	Procyon			✓	2,178	110
$P_g K$	ProGuard + Krakatau		✓	✓	2,178	110
$P_g P_c$	ProGuard + Procyon		✓	✓	2,178	110
AK	Artifice + Krakatau	✓		✓	2,178	110
AP_c	Artifice + Procyon	✓		✓	2,178	110
$AP_g K$	Artifice + ProGuard + Krakatau	✓	✓	✓	2,178	110
$AP_g P_c$	Artifice + ProGuard + Procyon	✓	✓	✓	2,178	110

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 in order of the six research questions.

5.1 RQ1: Performance Comparison

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

The results for this research question are collected from the experimental Scenario 1 (pervasive modifications) and Scenario 2 (reused boiler-plate code).

5.1.1 Pervasively Modified Code

A summary of the tools' performances and their optimal configurations on the generated data set are listed in Table 7. 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 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 has an impact on the performance of the tools. For example, ccfx gives a higher F-score when measuring similarity at character level than at line or token level. We present only the results for the best granularity level in each case here. The complete results of the tools can be downloaded from the study website (Ragkhitwetsagul and Krinke 2017a), including the generated data set before and after compilation/decompilation.

In terms of accuracy and F-score, the token-based clone detector ccfx is ranked first. The top 10 tools with highest F-score include ccfx (0.9760) followed by fuzzy-wuzzy (0.876), jplag-java (0.8636), 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. ccfx performs the best – possibly due to a combination of using a suffix tree matching algorithm on a small number of tokens (b=5). This means that ccfx performs similarity computation on one small chunk of code at a time. This approach is flexible and effective in handling code with pervasive modifications that spread changes over the whole file. We also manually investigated the similarity reports of poorly performing iclones and nicad and found that the tools were susceptible to code changes involving the two decompilers, Krakatau and Procyon. When comparing files after decompilation by Krakatau to Procyon with or without bytecode obfuscation, they could not find any clones and hence reported zero similarity.

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 expected since the tool is intended for plagiarism detection on

Table 7 Generated data set (Scenario 1): rankings (R) by F-scores (F1) and optimal configuration of every tool and technique

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
Clone det.										
ccfx (C)[a]	b=5,t=11	36	24	24	0.9952	0.9760	0.9760	0.9995	0.9760	1
deckard (T)[a]	mintoken=30 stride=2 similarity=0.95	17	44	227	0.9729	0.9461	0.7730	0.9585	0.8509	6
iclones (L)[a]	minblock=10 minclone=50	0	36	358	0.9196	0.9048	0.4886	0.7088	0.6345	27
nicad (L)[a]	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)[a]	threshold=4 ignoreVariableNames	5	150	165	0.9685	0.8477	0.8350	0.9262	0.8413	9
Plag. 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										
bsdif[a]	N/A	71	199	577	0.9224	0.6801	0.4230	0.8562	0.5216	30
diff (C)[a]	N/A	8	626	184	0.9190	0.5659	0.8160	0.9364	0.6683	26
difflib	whitespace=false autojunk=false	28	12	232	0.9756	0.9846	0.7680	0.9412	0.8629	4
fuzzywuzzy	token_set_ratio	85	58	176	0.9766	0.9342	0.8240	0.9772	0.8757	2
jellyfish	jaro_distance	78	340	478	0.9182	0.6056	0.5220	0.8619	0.5607	29

Table 7 (continued)

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
ngram	N/A	49	110	224	0.9666	0.8758	0.7760	0.9410	0.8229	22
cosine	N/A	48	292	458	0.9250	0.6499	0.5420	0.9113	0.5911	28

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

natural text rather than source code. Compression tools show promising results using 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. Other techniques for code similarity offer varied performance. Tools such as ngram, diff, cosine, jellyfish and bsdiff perform badly. They are ranked amongst the last positions at 22th, 26th, 28th, 29th, and 30th 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 from 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 7 include the ten highest AUC valued tools. We can see from the figure that ccfx is again the best performing tool with the highest AUC (0.9995), followed by 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 difflib. The lowest false negatives is given by diff. However, considering the large amount of false positive for diff (8,810 false positives 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.
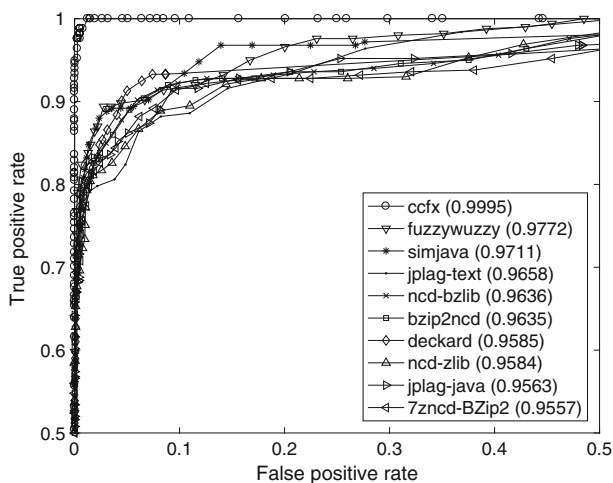

Fig. 7 The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC)

Compared to our previous study (Ragkhitwetsagul et al. 2016), we expanded the generated data set to be two times bigger. Although half (i.e. 50 files) of the generated data set are the same Java files as in the previous study, our 50 newly added files potentially introduce more diversity into the data set. This, as a result, makes our results more generalisable, i.e. mitigates our threats to external validity.

To sum up, we 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 amongst the last. However, we found that Python difflib and fuzzywuzzy perform surprisingly better with this expanded version of the data set than on the original data set in our previous study (Ragkhitwetsagul et al. 2016). They are both ranked highly amongst the top 5. Lastly, ccfx performed well on both the smaller data set in our previous study and the current data set, and is ranked the 1st on several error measures.

5.1.2 Boiler-plate Code

We report the complete evaluation of the tools on the SOCO data set with the optimal configurations in Table 8. Amongst the 30 tools, the top ranked tool in terms of F-score is jplag-text (0.9692), followed by simjava (0.9682), simian (0.9593) and jplag-java (0.9576). Most of the tools and techniques perform well on this data set. We observed high accuracy, precision, recall, and an F-score of over 0.7 for every tool except for diff and bsdiff. Since the data set contains source code that is copied and pasted with local modifications, the three clone detectors; ccfx, deckard, nicad, and simian; and plagiarism detectors; jplag-text, jplag-java and simjava; performed very well with F-scores between 0.9576 and 0.9692. ccfx and deckard produced the highest F-score when measuring similarity at character and token levels respectively. Other clone detectors including iclones, nicad, and simian provide the highest F-score at line level. The Python difflib and fuzzywuzzy are outliers of the Others group offering high performance against boiler-plate code with F-score of 0.9338 and 0.9443. Once again, these two string similarity techniques show promising results. The compression-based techniques are amongst the last although they still offer relatively high F-scores ranging from 0.8630 to 0.8776.

Regarding the overall performance over similarity thresholds of 0 to 100, the results are illustrated as ROC curves in Fig. 8. The tool with the highest AUC is difflib (0.9999), followed by sherlock (0.9996), fuzzywuzzy (0.9989), and simjava (0.9987).

To sum up, we observed that almost every tool detected boiler-plate code effectively by reporting high scores on all error measures. jplag-text, simjava, simian, jplag-java, and deckard are the top 5 tools for this data set in terms of F-score. Similar to pervasive modifications, we found the string matching techniques difflib and fuzzywuzzy ranked amongst the top 10.

5.1.3 Observations of the Tools' Performances on the Two Data Sets

We can notice a clear distinction between the F-score rankings of clone/plagiarism detectors and string/compression-based tools on the SOCO data set. This is due to the nature of boiler-plate code that has local modifications, contained within a single method or code block on which clone and plagiarism detectors perform well. However, on a more challenging pervasive modifications data set, there is no clear distinction in terms of ranking between dedicated code similarity techniques, compression-based, and general text similarity tools.

Table 8 SOCO data set (Scenario 3): rankings (R) by F-scores (F1) and optimal configuration of every tool and technique

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
Clone det.										
ccfx (C)[a]	b=15,16,17 t=12	25	42	15	0.9992	0.9125	0.9669	0.9905	0.9389	7
deckard (T)[a]	mintoken=50 stride=2 similarity=1.00	19	27	17	0.9993	0.9417	0.9625	0.9823	0.9520	5
iclones (L)[a]	minblock=40 minclone=50	19	20	57	0.9989	0.9519	0.8742	0.9469	0.9114	12
nicad (L)[a]	UPI=0.30 minline=5 rename=consistent abstract=condition	22	19	51	0.9990	0.9549	0.8874	0.9694	0.9199	9
simian (L)[a]	threshold=4 ignoreVariableNames	26	20	17	0.9994	0.9561	0.9625	0.9921	0.9593	3
Plag. det.										
jplag-java	t=12	29	26	13	0.9994	0.9442	0.9713	0.9895	0.9576	4
jplag-text	t=9	32	16	12	0.9996	0.9650	0.9735	0.9939	0.9692	1
plaggie	M=14	33	36	37	0.9989	0.9204	0.9183	0.9753	0.9193	10
sherlock	N=5, Z=0	22	22	54	0.9989	0.9477	0.8808	0.9996	0.9130	11
simjava	r=25	46	18	11	0.9996	0.9607	0.9757	0.9987	0.9682	2
simtext	r=12	17	73	19	0.9986	0.8560	0.9581	0.9887	0.9042	13
Compression										
7zncd-BZip2	mx=1,3,5	64	24	118	0.9979	0.9331	0.7395	0.9901	0.8251	26
7zncd-Deflate	mx=7	64	27	97	0.9982	0.9295	0.7859	0.9937	0.8517	24
7zncd-Deflate64	mx=7	64	27	96	0.9982	0.9297	0.7881	0.9957	0.8530	23
7zncd-LZMA	mx=7,9	69	11	99	0.9984	0.9699	0.7815	0.9940	0.8655	20
7zncd-LZMA2	mx=7,9	69	11	99	0.9984	0.9699	0.7815	0.9939	0.8655	20
7zncd-PPMd	mx=9	68	19	106	0.9981	0.9481	0.7660	0.9948	0.8474	25
bzip2ncd	C=1,2,3,..,8,9	54	20	94	0.9983	0.9473	0.7925	0.9944	0.8630	22
gzipncd	C=9	54	25	82	0.9984	0.9369	0.8190	0.9961	0.8740	16
icd[b]	ma=LZMA mx=1,3	84	12	151	0.9976	0.9618	0.6667	0.9736	0.7875	27
ncd-zlib	N/A	57	10	91	0.9985	0.9731	0.7991	0.9983	0.8776	14
ncd-bzlib	N/A	52	30	82	0.9983	0.9252	0.8190	0.9943	0.8689	18
xzncd	2,3 6,7,8,9,e	64 65	13	94	0.9984	0.9651	0.7925	0.9942	0.8703	17
Others										
bsdiff	N/A	90	2125	212	0.9652	0.1019	0.5320	0.9161	0.1710	29
diff (C)	N/A	29	7745	5	0.8845	0.0547	0.9890	0.9180	0.1036	30
difflib	autojunk=true whitespace=true	42	30	21	0.9992	0.9351	0.9536	0.9999	0.9443	6
fuzzywuzzy	ratio	65	30	30	0.9991	0.9338	0.9338	0.9989	0.9338	8
jellyfish	jaro_distance	82	0	162	0.9976	1.0000	0.6424	0.9555	0.7823	28
ngram	N/A	59	20	84	0.9984	0.9486	0.8146	0.9967	0.8765	15
cosine	N/A	68	50	68	0.9982	0.8851	0.8499	0.9973	0.8671	19

[a] — Tools that do not report similarity value directly. Similarity is measured at the granularity level of line (L), token (T), or character (C)
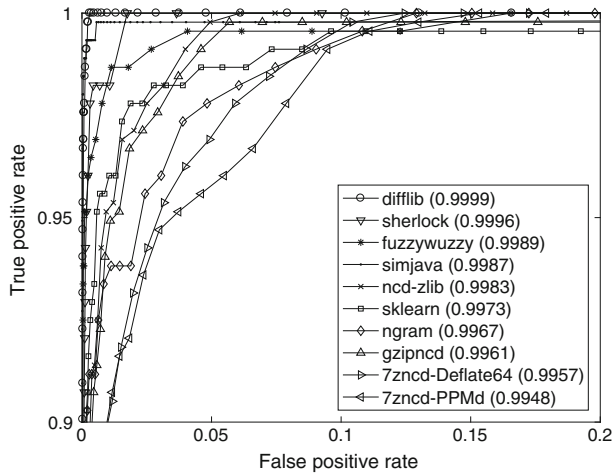
Fig. 8 The (zoomed) ROC curves of the 10 tools that have the highest area under the curve (AUC) for SOCO

We found that Python difflib string matching and Python fuzzywuzzy token similarity techniques even outperform several clone and plagiarism detection tools on both data sets. Provided that they are simple and easy-to-use Python libraries, one can adopt these two techniques to measure code similarity in a situation where dedicated tools are not available (e.g. unparsable, incomplete methods or code blocks). Compression-based techniques are not ranked at the top in either scenario, possibly due to the small size of the source code – NCD is known to perform better with large files.

5.2 RQ2: Optimal Configurations

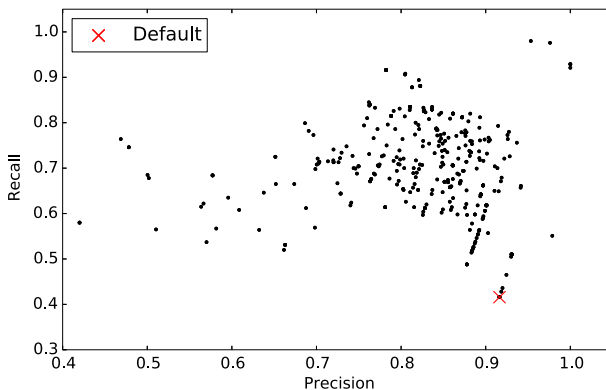*What are the best parameter settings and similarity thresholds for the techniques?*

Fig. 9 Trade off between precision and recall for 392 ccfx parameter settings. The default settings provide high precision but low recall against pervasive code modifications

Table 9 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

In the experimental Scenarios 1 and 2, we thoroughly analysed various configurations of every tool and found that some specific settings are sensitive to pervasively modified and boiler-plate code whilst others are not.

5.2.1 Pervasively Modified Code

The complete list of the best configurations of every tool for pervasive modifications from Scenario 1 can be found in the second column of Table 7. 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 reduce chances of finding 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 Fig. 9, we can see that the default settings of ccfx, b=50 and t=12 (denoted with a × symbol), provides a decent precision but very low recall. Whilst there is no setting for ccfx to obtain the optimal precision and recall at the same time, there are a few cases that ccfx can obtain high precision and recall as shown on the top right corner of Fig. 9. Our derived ccfx's optimal configuration is one of them. The best settings for precision and recall of ccfx are described in Table 9. 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 Fig. 10. 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.

5.2.2 Boiler-Plate Code

For boiler-plate code, we found another set of optimal configurations for the 30 tools by once again analysing a large search space of their configurations. The complete list of the best configurations for every tool from Scenario 3 can be found in the second column of Table 8. Similar to the generated data set, the derived optimal configurations for SOCO are different from the tools' default configurations. For example, ccfx's best configurations have a smaller b, minimum number of tokens, of 15 compared to the default value of 50
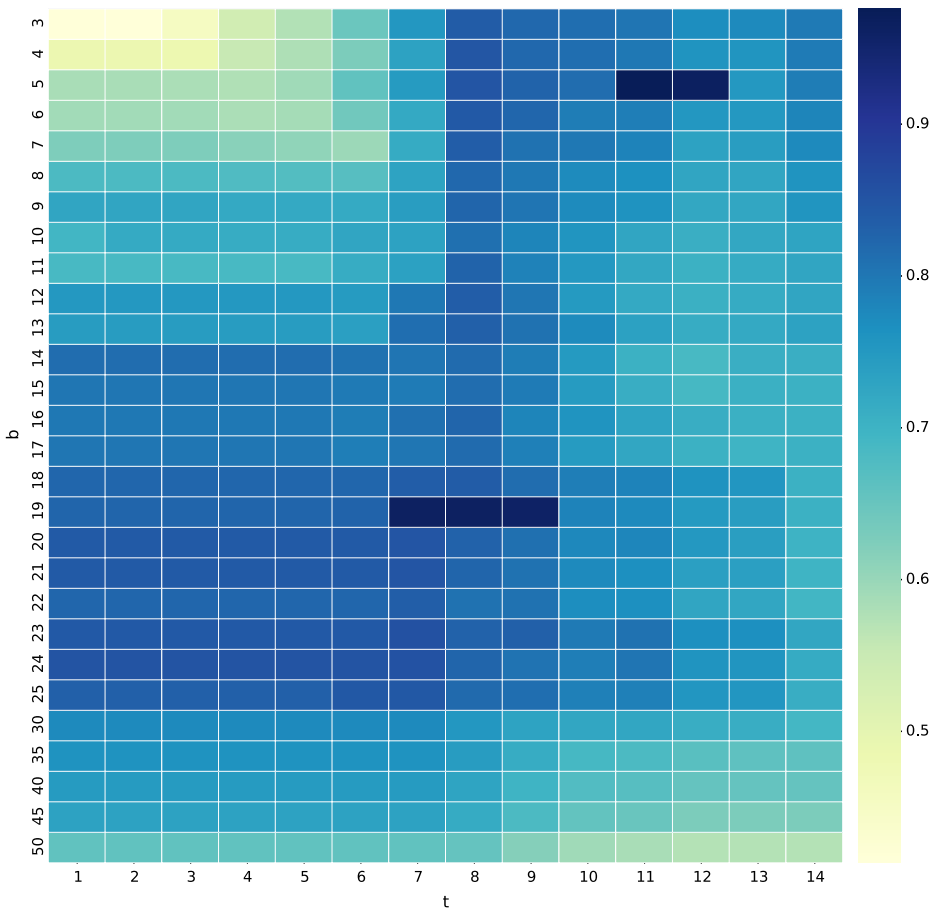
Fig. 10 F-scores of 392 ccfx's *b* and *t* parameter values on pervasive code modifications

whilst jplag-java's best configurations have a higher *t* value, the minimum number of tokens, of 12 compared to the default value of 9.

The results for both pervasively modified code and boiler-plate code show that the default configurations cannot offer the tools' their best performance. These empirical results support the findings of Wang et al. (2013) that one cannot rely on the tools' default configurations. We suggest researchers and practitioners try their best to tune the tools before performing any benchmarking or comparisons of the tools' results to mitigate the threats to internal validity in their studies. Our optimal configurations can be used as guidelines for studies involving pervasive modifications and boiler-plate code. Nevertheless, they are only effective against their respective data set and not guaranteed to work well on other data sets.

5.3 RQ3: Normalisation by Decompilation

How much does compilation followed by decompilation as a pre-processing normalisation method improve detection results of pervasively modified code?
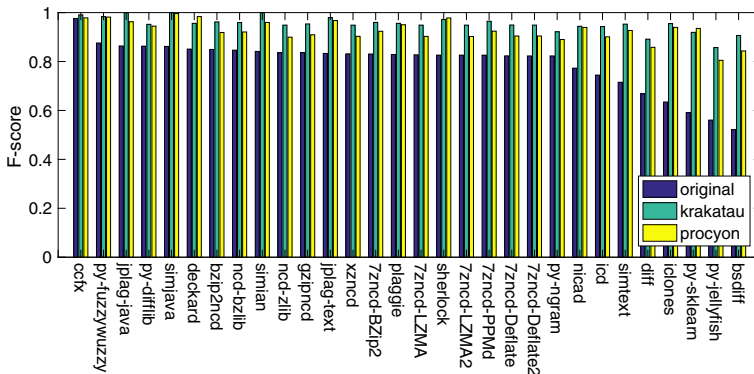
Fig. 11 Comparison of tool performances (F1-score) before and after decompilation

The results after adding compilation and decompilation for normalisation to the post-processing step before performing similarity detection on the generated data set in the experimental scenario 3 is shown in Fig. 11. We can clearly observe that decompilation by both Krakatau and Procyon boosts the F-scores of every tool in the study.

Table 10 shows the performances of the tools after decompilation by Krakatau in terms of false positive (FP) rate, false negative (FN) rate, 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 its number of false positives and negatives greatly reduced and three tools, simian, jplag-java, and simjava, even no longer report any false results. All compression or other techniques still report some false results. This supports the results of our previous study (Ragkhitwetsagul et al. 2016) that using compilation/decompilation as a code normalisation method can improve the F-scores of every tool.

To strengthen the findings, we performed a statistical test to see if the performances before and after normalisation via decompilation differ with statistical significance. We chose the non-parametric two-tailed Wilcoxon signed-rank test (Wilcoxon 1945)[5] and performed the test with a confidence interval value of 95% (i.e. $\alpha \leq 0.05$). Table 11 shows that the observed F-scores before and after decompilation are different with statistical significance for both Krakatau and Procyon. We complemented the statistical test by employing a non-parametric effect size measure called Vargha and Delaney's A_{12} measure (Vargha and Delaney 2000) to measure the level of differences between two populations. We choose Vargha and Delaney's A_{12} measure because it is robust with respect to the shape of the distributions being compared (Thomas et al. 2014). Put it another way, it does not require the two populations under comparison to be normally distributed, which is the case in our results of the tools' F1 scores. According to Vargha and Delaney (2000), the A_{12} value of 0.5 means there is no difference between the two populations. A_{12} value over or below 0.5 means the first population outperforms the second population, and vice versa. The guideline in Vargha and Delaney (2000) shows that 0.56 is interpreted as small, 0.64 as medium, and 0.71 as large. Using this scale, our F-score differences after decompilation by Krakatau ($A_{12} = 0.969$) and Procyon ($A_{12} = 0.937$) compared to the original are large. According to the interpretation of A_{12} in Vargha and Delaney (2000), with Krakatau's A_{12} of 0.969 we

---

[5]However, we also tried using the randomisation (i.e. permutation) test (Fisher 1935; Box et al. 1978) on the results and found identical test results in all cases

Table 10 Optimal configuration of every tool obtained from the generated[decomp] 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[ab] (T)	b=5, t=8	50	0	18	0.9982	1.0000	0.9820	0.9991	0.9909	4
deckard[ab] (L)	mintoken=30	29	0	84	0.9916	1.0000	0.9160	0.9459	0.9562	11
	stride=1									
	similarity=0.95									
iclones[a] (L)	minblock=8	10	0	86	0.9914	1.0000	0.9140	0.9610	0.9551	14
	minclone=50									
nicad[ab] (T)	UPI=0.30	19	0	106	0.9894	1.0000	0.8940	0.9526	0.9440	24
	minline=8									
	rename=blind									
	abstract=literal									
simian[ab] (T)	threshold=3	17	0	0	1.0000	1.0000	1.0000	0.9960	1.0000	1
	ignoreidentifiers									
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[b]	r=18	17	0	0	1.0000	1.0000	1.0000	0.9998	1.0000	1
simtext	r=4;	33	33	60	0.9907	0.9661	0.9400	0.9862	0.9529	16
	r=5	31								
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[b]	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
difflib	autojunk=true	23	28	66	0.9906	0.9709	0.9340	0.9823	0.9521	17
fuzzywuzzy	token_set_ratio	90	0	32	0.9968	1.0000	0.9680	0.9966	0.9837	5
jellyfish	jaro_winkler	89	40	220	0.9740	0.9512	0.7800	0.9473	0.8571	30

Table 10 (continued)

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
ngram	N/A	60	48	104	0.9848	0.9492	0.8960	0.9726	0.9218	26
cosine	N/A	68	98	66	0.9836	0.9050	0.9340	0.9955	0.9193	27

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

[b] —Tools that have several optimal configurations. The complete lists can be found on our study website

can compute the probability that a random X_1 score from the set of tools' performance after decompilation by Krakatau will be greater than a random X_2 score from the set of tools' performance before decompilation by $2A_{12} - 1 = 2 \times 0.969 - 1 = 0.938$. This A_{12} effect size confirms that the tools' performance after decompilation by Krakatau will be higher than the original 93.8% of the time. The similar finding also applies to Procyon (87.4%). The large effect sizes clearly supports the findings that compilation and decompilation is an effective normalisation technique against pervasive modifications.

To gain insight, 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 Fig. 4 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 with respect to minor obfuscations and the transformations made by Artifice and Pro-Guard 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 10 and separately reported the complete list of optimal configurations on our study website (Ragkhitwetsagul and Krinke 2017a).

Normalisation via decompilation using Procyon also improves the performance of the similarity detectors, but not as much as Krakatau (see Table 12). Interestingly, Procyon performs slightly better for deckard, sherlock, and cosine. An example of code before and after decompilation by Procyon is shown in Fig. 4 at the bottom.

The main difference between Krakatau and Procyon is that Procyon attempts to produce much more high-level source code whilst 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.

5.4 RQ4: Reuse of Configurations

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

Table 11 Wilcoxon signed-rank test of tools' performances before and after decompilation by Krakatau and Procyon ($\alpha = 0.05$)

Test	p-value	Significant?	Effect size (A_{12})
Before-after decompiled by Krakatau	1.863e-09	Yes	0.969 (large)
Before-after decompiled by Procyon	1.863e-09	Yes	0.937 (large)

Table 12 Optimal configuration of every tool obtained from the generated[decomp] 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[a] (L)	b=20, t=1..7	11	4	38	0.9958	0.9959	0.962	0.9970	0.9786	4
deckard[a] (T)	mintoken=30 stride=1, inf similarity=1.00	10	0	32	0.9968	1.0000	0.9680	0.9978	0.9837	2
iclones[a] (C)	minblock=10 minclone=50	0	18	98	0.9884	0.9804	0.9020	0.9508	0.9396	11
nicad[a] (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[a] (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 r=default	15 0	42	100	0.9858	0.9554	0.9000	0.9686	0.9269	14
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
difflib	autojunk=true	26	12	94	0.9894	0.9869	0.9060	0.9788	0.9447	10
fuzzywuzzy	token_set_ratio	90	0	36	0.9964	1.0000	0.9640	0.9992	0.9817	3

Table 12   (continued)

Tool	Settings	T	FP	FN	Acc	Prec	Rec	AUC	F1	R
jellyfish	jaro_winkler	87	84	270	0.9646	0.8968	0.7300	0.9218	0.8049	30
ngram	N/A	58	8	192	0.9800	0.9902	0.8080	0.9714	0.8899	27
cosine	N/A	69	54	74	0.9872	0.9449	0.9260	0.9897	0.9354	12

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

We answer this research question using the results from RQ1 and RQ2 (experimental Scenario 1 and 2 respectively). For the 30 tools from RQ1, we applied the derived optimal configurations obtained from the generated data set (denoted as C_{gen}) to the SOCO data set. Table 13 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. We noticed that the low F-scores when C_{gen} are reused on SOCO come from high number of false positives possibly due to their relaxed configurations.

To confirm this, we refer for the best configurations (settings and threshold) for the SOCO data set discussed in RQ1 (see Table 8), the comparison of best configurations between the two data sets is shown in Table 13. 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

Table 13 The table displays the results after applying the best configurations (C_{gen}) from Scenario 1 to the SOCO data set and the derived best configurations for the SOCO set (C_{soco}). The selected 10 tools are compared by their F-scores

Tools	C_{gen}				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
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
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	46	0.9682
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

dataset-based optimal configurations, C_{soco}, to be very different from the configuration for the generated data set C_{gen}. Although the table shows only the top 10 tools from the generated data set, the same findings apply for every tool in our study. The complete results can be found from our study website (Ragkhitwetsagul and Krinke 2017a).

Lastly, we noticed that the best thresholds for the tools are very different between one data set and 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.

5.5 RQ5: Ranked Results

Which tools perform best when only the top n results are retrieved?

In experimental scenario 4, we applied 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.

5.5.1 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 14, we only reported the *n* equals to 1,000 since it is the number of true positives in the data set. The tools' optimal configurations are not included in the table but can be found on our study website (Ragkhitwetsagul and Krinke 2017a). The ccfx tool is ranked 1st with the highest prec@n of 0.976 followed by simjava, and fuzzywuzzy. In comparison with the rankings for F-scores, the ranking of the ten tools changed slightly, as simjava and simian perform better whilst jplag-java and difflib tool now

Table 14 Top-10 rankings of using prec@n, ARP, and MAP over the generated data set with the tools' optimal configurations

| Rank	Pair-based		Query-based	
	F-score	prec@n	ARP	MAP
1	(0.976) ccfx	(0.976) ccfx	(1.000) ccfx	(1.000) ccfx
2	(0.876) fuzzywuzzy	(0.860) simjava	(0.915) fuzzywuzzy	(0.949) fuzzywuzzy
3	(0.864) jplag-java	(0.858) fuzzywuzzy	(0.913) ncd-bzlib	(0.943) ncd-bzlib
4	(0.863) difflib	(0.842) simian	(0.912) 7zncd-BZip2	(0.942) bzip2ncd
5	(0.862) simjava	(0.836) deckard	(0.909) bzip2ncd	(0.938) 7zncd-BZip2
6	(0.851) deckard	(0.836) jplag-java	(0.900) 7zncd-PPMd	(0.937) gzipncd
7	(0.849) bzip2ncd	(0.832) bzip2ncd	(0.900) gzipncd	(0.935) ncd-zlib
8	(0.847) ncd-bzlib	(0.828) difflib	(0.898) ncd-zlib	(0.933) jplag-text
9	(0.841) simian	(0.826) ncd-bzlib	(0.898) xzncd	(0.930) 7zncd-PPMd
10	(0.836) ncd-zlib	(0.820) 7zncd-BZip2	(0.895) 7zncd-LZMA2	(0.929) xzncd
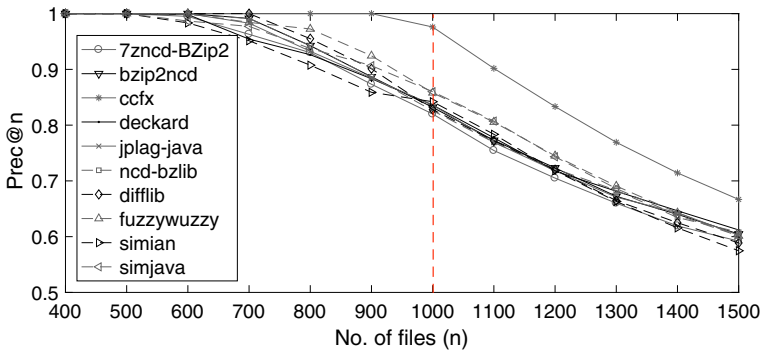
Fig. 12 Precision-at-n of the tools according to varied numbers of *n* against generated data set

performed worse. ncd-zlib is no longer in the top 10 and is replaced by 7ncd-BZip2 in the 10th place.

As illustrated in Fig. 12, varying fifteen *n* values of prec@n 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 1.0 until the top 500 pairs. However, at the top 600 pairs, the performance of 7zncd-BZip2, deckard, ncd-bzlib, simian and simjava started dropping. bzip2ncd, jplag-java, and fuzzywuzzy started reporting false positives after the top 700 pairs whilst 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. Due to a fairly small data set, this finding of perfect 1.0 prec@n until the first 500 pairs may not generalise to other data sets, as the similar performances achieved by the tools on the first 500 pairs might be due to intrinsic properties of the analysed programs.

For the SOCO data set, we varied the *n* value from 100 to 800, also stepping up by 100. The results in Table 15 used the *n* value of 453 which is the number of true positives in the corrected ground truth. We can clearly see that the ranking of 10 tools using prec@n closely resembles the one using F-scores. jplag-text is the top ranked tool followed by simjava, jplag-java, simian, and deckard. The ranking of eight tools is exactly the same as using F-score. jplag-java and nicad perform slightly worse using prec@453 and move down one position. The overall performances of the tools across various *n* values is depicted in Fig. 13 with the dotted line representing the number of true positives. The chart is somewhat analogous to the generated data set (Fig. 12). Most of the tools started reporting false positives at the top 300 pairs except jplag-java, difflib, fuzzywuzzy and simjava. After the top 400 pairs, no tool could any longer maintain 100% true positive results.

Since prec@n is calculated from a set of top-*n* ranked results, its value shows how fast a tool can retrieve correct answers to a limited set of *n* most similar files. It also reflects how well the tool can differentiate between similar and dissimilar documents. A good tool should not be confused and should produce a large gap in the similarity values between the true positive and the true negative results. In this study, ccfx and jplag-text have shown to be the best tools in terms of prec@n for pervasive modifications and boiler-plate code respectively. They are the also the best tools based on F-scores in RQ1.

Table 15 Top-10 rankings of using prec@n, ARP, and MAP over the SOCO data set with the tools' optimal configurations

Rank	Pair-based		Query-based	
	F-score	prec@n	ARP	MAP
1	(0.969) jplag-text	(0.965) jplag-text	(0.998) jplag-java	(0.997) jplag-java
2	(0.968) simjava	(0.960) simjava	(0.998) difflib	(0.997) difflib
3	(0.959) simian	(0.956) simian	(0.989) ccfx	(0.993) jplag-text
4	(0.958) jplag-java	(0.947) deckard	(0.989) simjava	(0.988) simjava
5	(0.952) deckard	(0.943) jplag-java	(0.987) gzipncd	(0.987) gzipncd
6	(0.944) difflib	(0.938) difflib	(0.986) jplag-text	(0.987) ncd-zlib
7	(0.939) ccfx	(0.929) ccfx	(0.985) ncd-zlib	(0.986) sherlock
8	(0.934) fuzzywuzzy	(0.929) fuzzywuzzy	(0.984) 7zncd-Deflate	(0.986) 7zncd-Deflate64
9	(0.920) nicad	(0.914) plaggie	(0.984) 7zncd-Deflate64	(0.986) 7zncd-Deflate
10	(0.919) plaggie	(0.901) nicad	(0.983) 7zncd-LZMA	(0.984) fuzzywuzzy

5.5.2 Average r-Precision

ARP is a query-based error measure that needs knowledge of ground truth. Since we knew the ground truth for our two data sets, we did not need to vary the values of n as in prec@n. The value of 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 10). We cut off after 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 14. We can see that ccfx is still ranked first with the perfect ARP of 1.0000 followed by fuzzywuzzy. ncd-bzlib now performs much better using ARP and is ranked third. Interesting, the 3rd to 10th ranks are all compression-based tools. This shows that with the presence of pervasive modifications, code similarity using NCD-compression method is better at query-based results than most of the clone and plagiarism detectors and the string similarity tools.
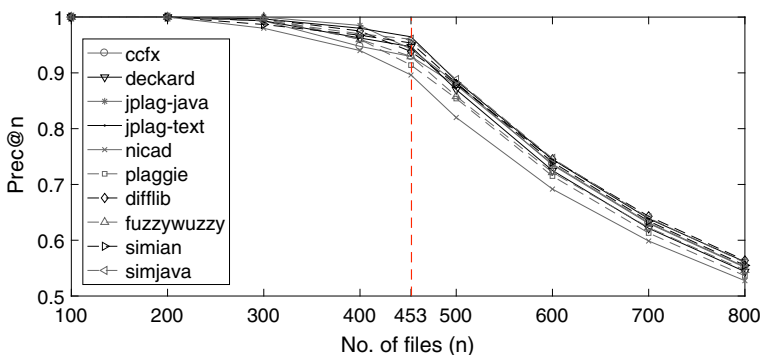

Fig. 13 Precision-at-n of the tools according to varied numbers of n against SOCO data set

Table 16 One-tailed randomisation test with 100K samples of the ARP and MAP values from the generated data set

Tool	ccfx	fuzzywuzzy	ncd-bzlib	bzip2ncd	ncd-zlib	deckard	simjava	jplag-java	simian	difflib
ccfx		►	►	►	►	►	►	►	►	►
fuzzywuzzy	□		□	□	□	□	□	□	□	□
ncd-bzlib	□	□		□	□	□	□	□	□	□
bzip2ncd	□	□	□		□	□	□	□	□	□
ncd-zlib	□	□	□	□		□	□	□	□	□
deckard	□	□	□	□	□		□	□	□	□
simjava	□	□	□	□	□	□		□	□	□
jplag-java	□	□	□	□	□	□	□		□	□
simian	□	□	□	□	□	□	□	□		□
difflib	□	□	□	□	□	□	□	□	□	

► — statistically significant difference of 1st tool's ARP (row) to $2nd^{nd}$ tool's ARP (column), i.e. $\alpha \leq 0.05$

□ — no statistically significant difference

For SOCO, only files with known, 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 value of r here was not fixed as for the generated data set. It depended on how many relevant answers existed 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 15. jplag-java and difflib are ranked first with an ARP of 0.998, followed by ccfx and simjava both with an ARP of 0.989. Similar to the findings for the generated data set, compression-based tools work well with a query-based approach by having 5 NCD tools ranked in the top 10.

Since ARP are computed based on means, we performed a statistical test to strengthen our results by testing for the statistical significance of differences in the set of r-precision values between tools. We chose a one-tailed non-parametric randomisation test (i.e. permutation test) due to its robustness in information retrieval as shown by Smucker et al. (2007).[6] We performed the test using 100,000 random samples with a confidence interval value of 95% (i.e. $\alpha \leq 0.05$). The statistical test results are shown in Tables 16 and 17. The tables are matrices of pairwise one-tailed statistical test results in the direction of rows $\geq$ columns. The symbol ► represents statistical significance whilst the symbol □ represents no statistical significance. For example, in Table 16, the ► on the left most of the top row [ccfx, fuzzywuzzy] shows that the mean of r-precision values of ccfx are higher than or equal to fuzzywuzzy's with statistical significance. On the other hand, we can see that the mean of r-precision values of fuzzywuzzy is higher than ncd-bzlib with no statistical significance as represented by □ at the location of [fuzzywuzzy, ncd-bzlib].

For the generated data set (Table 16), we found that ccfx is the only tool that dominates other tools on their r-precisions values with statistical significance. For SOCO data set (Table 17), jplag-java and difflib outperform 7zncd-Deflate, 7zncd-Deflate64, and 7zncd-LZMA with statistical significance.

---

[6]We also tried using one-tailed Wilcoxon signed-rank test on the results and found identical test results in all cases.

Table 17 One-tailed randomisation test with 100K samples of the ARP values from the SOCO data set

Tool	jplag-java	difflib	ccfx	simjava	gzipncd	jplag-text	ncd-zlib	deflate	deflate64	LZMA
jplag-java		☐	☐	☐	☐	☐	☐	▶	▶	▶
difflib	☐		☐	☐	☐	☐	☐	▶	▶	▶
ccfx	☐	☐		☐	☐	☐	☐	☐	☐	☐
simjava	☐	☐	☐		☐	☐	☐	☐	☐	☐
gzipncd	☐	☐	☐	☐		☐	☐	☐	☐	☐
jplag-text	☐	☐	☐	☐	☐		☐	☐	☐	☐
ncd-zlib	☐	☐	☐	☐	☐	☐		☐	☐	☐
deflate	☐	☐	☐	☐	☐	☐	☐		☐	☐
deflate64	☐	☐	☐	☐	☐	☐	☐	☐		☐
LZMA	☐	☐	☐	☐	☐	☐	☐	☐	☐	

▶ — statistically significant difference of 1st tool's ARP (row) to $2nd^{nd}$ tool's ARP (column), i.e. $\alpha \leq 0.05$

☐ — no statistically significant difference

ARP tells us how well the tools perform when we want all the true positive results in a query-based manner. For example, in automated software repair one wants to find similar source code given some original, buggy, source code that one possesses. One can use the original source code as a query and look for similar source files in a set of source code files. In our study, ccfx is the best tool for this retrieval method against pervasive modifications. jplag-java and difflib are the best tool for boiler-plate code.

5.5.3 Mean Average Precision

We included 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 for 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 as in the ARP calculations for the generated and the SOCO data set. The results for MAP are reported in Tables 14 and 15.

For the generated data set (Table 14), the rankings are very similar to those for ARP. ccfx, fuzzywuzzy, and ncd-bzlib are ranked 1st, 2nd and 3rd. For SOCO (Table 15), the rankings are very different to those obtained when using F-score and prec@n but similar to those for ARP. Tools jplag-java and difflib become the best performers followed by jplag-text and simjava.

Compression-based tools are again found to offer good performance with MAP. Five tools are ranked in the top 10 for both the generated and boiler-plate code data sets.

Similarly, since MAP is also computed based on mean, we performed a one-tailed non-parametric randomisation statistical test on pairwise comparisons of the tools' MAP values. The test results are shown in Tables 16 and 18. For the generated data set, we found the same results of ccfx dominating other tools' MAPs with statistical significance. For the SOCO data set, we found that jplag-java and difflib outperform gzipncd, ncd-zlib, sherlock, 7zncd-Deflate64, 7zncd-Deflate, and fuzzywuzzy with statistical significance.

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

Table 18 One-tailed randomisation test with 100K samples of the MAP values from SOCO data set

Tool	jplag-java	difflib	jplag-text	simjava	gzipncd	ncd-zlib	sherlock	deflate64	deflate	fuzzywuzzy
jplag-java		☐	☐	☐	▶	▶	▶	▶	▶	▶
difflib	☐		☐	☐	▶	▶	▶	▶	▶	▶
jplag-text	☐	☐		☐	☐	☐	☐	☐	☐	☐
simjava	☐	☐	☐		☐	☐	☐	☐	☐	☐
gzipncd	☐	☐	☐	☐		☐	☐	☐	☐	☐
ncd-zlib	☐	☐	☐	☐	☐		☐	☐	☐	☐
sherlock	☐	☐	☐	☐	☐	☐		☐	☐	☐
deflate64	☐	☐	☐	☐	☐	☐	☐		☐	☐
deflate	☐	☐	☐	☐	☐	☐	☐	☐		☐
fuzzywuzzy	☐	☐	☐	☐	☐	☐	☐	☐	☐	

▶ — statistically significant difference of 1st tool's MAP (row) to 2nd tool's MAP (column), i.e. $\alpha \leq 0.05$

☐ — no statistically significant difference

of being query-based measure instead of a pair-based measure. It shows how well a tool performs on average when it has to find all true positives for each query. In this study, the best performing tool in terms of MAP is ccfx for pervasively modified code and jplag-java and difflib for boiler-plate code respectively.

5.6 RQ6: Local + Global Code Modifications

> *How well do the techniques perform when source code containing boiler-plate code has been pervasively modified?*

Using the results from Experimental Scenario 5, we present the tools' performances based on F-scores in Table 19 and show the distribution of F-scores in Fig. 14. The F-scores are grouped according to the 10 pervasive code modification types (see Table 6). The numbers are highlighted when F-scores are higher than 0.8.

5.6.1 Tools' Performances vs. Individual Pervasive Modification Type

On the original boiler-plate code without any modification (O), every tool except iclones, nicad, bsdiff, and diff report high F-scores ranging from 0.8 to 1.0. This shows that most tools with their default configurations do not have a problem detecting boiler-plate code. The tool nicad performed poorly, possibly due to default configurations that aim at clones without variable renaming and code abstraction at all (i.e. set renaming=none and abstract=none). iclone's default configurations of minimum 100 of clone tokens are too high compared to the optimal configurations of 40 found in RQ1. diff and bsdiff are too general to handle code with local modifications.

The tools perform worse after pervasive modifications are applied on top of the boiler-plate code. Source code obfuscation by Artifice (A) has strong effects to ccfx, iclones, nicad, simian, bsdiff, and diff according to low F-scores of 0.0 to 0.2. deckard, jplag-java, plaggie, simjava, difflib, fuzzywuzzy and ngram maintained their high F-scores of over 0.9. Interestingly, jplag-java reported a perfect F-score of 1.0 possibly due to it being designed for detecting plagiarised code which is usually pervasively modified at source code level.

Table 19 F-scores of the tools on SOCO[gen] using the default configurations (with optimised threshold). **Highlighted** values have F-score higher than 0.8

Tool	F-Score									
	O	A	K	P_c	P_g K	P_g P_c	A K	A P_c	A P_g K	A P_g P_c
Clone det.										
ccfx (C)[a]	**0.8911**	0.3714	0.0000	0.6265	0.0000	0.1034	0.0000	0.2985	0.0000	0.1034
deckard (T)[a]	**0.9636**	**0.9217**	0.1667	0.3333	0.0357	0.2286	0.1667	0.3252	0.0357	0.2286
iclones (L)[a]	0.5000	0.0000	0.0000	0.0357	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
nicad (T)[a]	0.5823	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
simian (L)[a]	**0.8350**	0.1034	0.0357	0.1356	0.0000	0.0357	0.0000	0.0357	0.0000	0.0357
Plagiarism det.										
jplag-java	**1.0000**	**1.0000**	0.7429	**0.9524**	0.2973	0.4533	0.7547	**0.9720**	0.2973	0.4507
jplag-text	**0.9815**	0.6265	0.5581	0.6304	0.3590	0.4250	0.4906	0.5581	0.3590	0.4304
plaggie	**0.9636**	**0.9159**	0.7363	**0.9372**	0.2171	0.4626	0.7363	**0.9423**	0.2171	0.4626
sherlock	**0.9483**	**0.8298**	0.7872	**0.8298**	0.3061	0.3516	0.6744	0.7826	0.3061	0.3516
simjava	**0.9649**	**0.9815**	**1.0000**	0.7525	0.3188	0.3913	**0.8041**	0.7525	0.3188	0.3913
simtext	**0.9649**	0.7191	0.1667	0.4932	0.0357	0.1667	0.0702	0.2258	0.0357	0.1667
Compression										
7zncd-BZip2	**0.9273**	0.7736	0.6852	**0.8649**	0.2446	0.3704	0.6423	0.7465	0.2446	0.3704
7zncd-Deflate	**0.9483**	0.7579	0.6935	**0.8406**	0.2427	0.3333	0.6360	0.7418	0.2427	0.3333
7zncd-Deflate64	**0.9483**	0.7579	0.6935	**0.8406**	0.2427	0.3333	0.6360	0.7373	0.2427	0.3333
7zncd-LZMA	**0.9649**	0.7967	0.7488	**0.8851**	0.2663	0.3842	0.6768	0.7665	0.2632	0.3842
7zncd-LZMA2	**0.9649**	0.7934	0.7536	**0.8851**	0.2718	0.3923	0.6700	0.7632	0.2697	0.4000
7zncd-PPMd	**0.9623**	0.7965	0.7628	**0.8909**	0.2581	0.3796	0.6667	**0.8019**	0.2581	0.3796
bzip2ncd	**0.9649**	**0.8305**	**0.8302**	**0.9273**	0.3590	0.4681	0.7612	**0.8448**	0.3562	0.4681
gzipncd	**0.9623**	0.7965	0.7628	**0.8909**	0.2581	0.3796	0.6667	**0.8019**	0.2581	0.3796
icd	**0.9216**	0.5058	0.4371	0.5623	0.2237	0.2822	0.3478	0.4239	0.2237	0.2822
ncd-zlib	**0.9821**	**0.8571**	**0.8246**	**0.9432**	0.4021	0.4920	0.7491	**0.8559**	0.3963	0.4920
ncd-bzlib	**0.9649**	**0.8269**	**0.8269**	**0.9273**	0.3529	0.4634	0.7500	**0.8448**	0.3500	0.4719
xzncd	**0.9734**	**0.8416**	0.7925	**0.9198**	0.3133	0.4615	0.7035	**0.8148**	0.3133	0.4615
Others										
bsdiff	0.4388	0.2280	0.1529	0.2005	0.1151	0.1350	0.1276	0.1596	0.1152	0.1353
diff (C)	0.2835	0.2374	0.1585	0.2000	0.1296	0.1248	0.1530	0.1786	0.1302	0.1249
difflib	**0.9821**	**0.9550**	**0.8952**	**0.9565**	0.4790	0.5087	**0.8688**	**0.9381**	0.4606	0.5091
fuzzywuzzy	**1.0000**	**0.9821**	**0.9259**	**0.9636**	0.4651	0.5116	**0.9074**	**0.9541**	0.4557	0.5116
jellyfish	**0.9273**	0.7253	0.6400	0.6667	0.2479	0.3579	0.5513	0.5000	0.2479	0.3662
ngram	**1.0000**	**0.9464**	**0.8952**	**0.9346**	0.4110	0.4490	**0.8785**	**0.8908**	0.4054	0.4578
cosine	**0.9074**	0.6847	0.7123	0.6800	0.3500	0.3596	0.5823	0.5287	0.3500	0.3596

[a] — A tool that does not report similarity value directly. The similarity is measured at the granularity level of line (L), token (T), or character (C)

According to the boxplot in Fig. 14, code after decompilation by Krakatau (K) results in lower F-scores than after decompilation by Procyon. Since the Krakatau decompilation process generates source code that is close to Java bytecode and mostly structurally different from the original, its generated code is challenging for tools that are based on lexical and syntactic similarity. In the group of clone detectors, ccfx, iclones, and nicad did not report any correct results at all (F-score = 0.0) whilst deckard and simian reported very low F-scores of 0.1667 and 0.0357 respectively. Code after decompilation by Procyon (P_c) had milder effects than Krakatau and Artifice. The tool fuzzywuzzy is the best for both K and P_c with F-scores of 0.9259 and 0.9636 respectively.

A combination of ProGuard and either Krakatau or Procyon ($P_g K$, $P_g P_c$) reported the lowest F-scores as can be clearly seen from Fig. 14. This is due to bytecode modifications (e.g. renaming classes, fields, and variables, package hierarchy flattening, class repackaging, merging classes and modifying package access permissions) performed by ProGuard combined with a decompilation process that greatly changed both the lexemes and the structure of the code. It is interesting to see that difflib and fuzzywuzzy, a token matching technique, are the highest performing tools with F-scores of 0.4790 and 0.5116 for $P_g K$ and $P_g P_c$ respectively. Thus, in the presence of pervasive modifications that heavily or completely change code structure, using a simpler, general, text similarity technique may give a higher chance of finding similar code than dedicated code similarity tools.

Code after source code obfuscation by Artifice and decompilation by Krakatau and Procyon (AK, AP_c) has comparable results to K and P_c with marginal differences. Fuzzywuzzy and jplag-java are the best tools for this modification type.

Lastly, two combinations of obfuscation and decompilation ($AP_g K$, $AP_g P_c$) also provide almost identical F-score results to $P_g K$ and $P_g P_c$. This suggests that the pervasive modifications made to source code obfuscation may be no longer effective if decompilation is included. Vice versa, the modifications made by bytecode obfuscation persist through the compilation and decompilation process. Difflib and fuzzywuzzy are the best tools for this modification type.

To sum up, we found that most of the tools perform well on detecting boiler-plate code, and report lower performance when adding pervasive modifications. Some clone detection tools can tolerate pervasive modifications made by source code obfuscators, but all are susceptible to pervasive changes made by decompilers or a combination of a bytecode obfuscator and decompilers. Plagiarism detectors offer decent results over the 10 modification types. Interestingly, fuzzywuzzy and difflib, token and string matching techniques, outperformed dedicated tools on heavily modified code with a combination of obfuscators and decompilers.
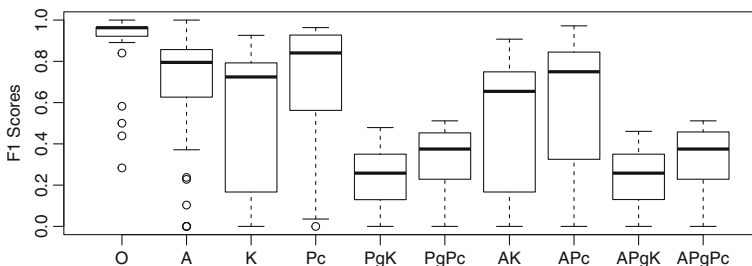

Fig. 14 Distribution of tools performance for each pervasive modification type

5.7 Overall Discussions

In summary, we have answered the six research questions after performing five 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. A comparison of the tools on boiler-plate code in the SOCO data set found the jplag-text plagiarism detector performed the best followed by simjava, simian, jplag-java, and deckard.

The experiment using compilation/decompilation for normalisation showed that compilation/decompilation is effective and improves similarity detection techniques with statistical significance. 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 biassed by the particular data set. One cannot get optimal results from tools by directly applying the optimal derived parameter settings and similarity thresholds for 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 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.

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 measurements 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, when one is interested in looking at only the top n most similar source code pairs due to limited time for manual inspection or when one uses a file to query for the other most similar files.

Lastly, we compare the tools on a data set of pervasively modified boiler-plate code. We found that whilst most tools offered high performance on boiler-plate code, they performed much worse after pervasive modifications were applied. We observed that pervasively modified code with changes made from a combination of bytecode obfuscation by ProGuard and the two decompilers had strongest effects on the tools' F-scores.

5.8 Threats to Validity

Construct 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. We investigated whether our obfuscators (Artifice and ProGuard), compiler (javac) and decompilers (Krakatau and Procyon) offer code modifications that are commonly found in code cloning and code plagiarism (see Table 1). However, they may not totally represent

all possible pervasive modifications found in software. 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.

Internal validity: Although we have attempted to use the tools with their best parameter settings, we cannot guarantee that we have done so successfully and it may be possible that the poor performance of some detectors is due to wrong usage as opposed to 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 were 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 completely representative of all available techniques and tools.

The generated (100 Java files) and SOCO (259 Java files) data sets are fairly small and contain a single class with one or a few methods. They might not adequately represent real software projects. Hence, our results are limited to pervasive modifications and boiler-plate code at a file-level, not a whole software project. The optimal configurations presented in this paper are found relative to the data set of code modifications from which they were derived and may not generalise to all type of code modifications. 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 affect both the originality of one's idea, one's credibility, and also the quality of one's 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 the similarity of two programs. Two programs can be similar at the level of purpose, algorithm, or implementation (Zhang et al. 2012). Most 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 whilst preserving its semantics (Whale 1990; Collberg et al. 1997, 2002, 2003). 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) whilst decompilers like Krakatoa (Proebsting and Watterson

1997), Krakatau (Grosse 2016) and Procyon (Strobel 2016) attempt to decompile even in the presence of obfuscation.

Although there are 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. (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. (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. (2010) compare five plagiarism detectors in term of their features and performances against 17 code modifications. Burd and Bailey (2002) compare five clone detectors for preventive maintenance task. Biegel et al. (2011) compare three code similarity measures to identify code that need refactoring. 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). (Svajlenko and Roy 2016) developed a clone evaluation framework called BigCloneEval that aimed to automatically measure clone detectors' recall on the BigCloneBench data set. The BigCloneBench's manually-confirmed clone oracle is built from IJaDataset, the repository of 25,000 Java open source projects, by searching for methods containing keywords and source code patterns representing 43 functionalities. Whilst BigCloneEval offers a benefit of manually confirmed clones from a large set of real-world Java software projects, their clone oracle, and also the measured recall, is limited to the selected functionalities. If a tool reports other clone pairs besides these 43 functionalities, the framework does not take them into account. Although our two data sets in this study are much smaller in size in comparison with the BigCloneBench, we were able to measure both precision and recall. Since we created one data set using code obfuscators, a compiler, and decompilers, and reused another data set from a competition, we had a complete knowledge of the ground truth for both of them and could take all possible similar code pairs, i.e. clones, into account.

Several code obfuscation methods can be found in the work of Luo et al. (2014). The techniques utilised include obfuscation by different compiler optimisation 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 performed by Ceccato et al. (2009). They evaluated how layout obfuscation by identifier renaming affects the participants' comprehension of, and ability to modify, two given programs. They found that obfuscation by identifier renaming could slow down an attack by two to four times the time needed for clear, un-obfuscated programs. Their later study (Ceccato et al. 2013) 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 decompiled obfuscated bytecode and compared the tools' performances based on the resulting source code.

There are studies that try to enhance the 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 mainly by increasing the 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 in 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 2017b).

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 optimisation 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.

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).

7 Conclusions

This study of 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 run against five 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. ccfx offers the highest performance on pervasively modified code and jplag-text on boiler-plate code. However, general string matching techniques, fuzzywuzzy and difflib, outperform dedicated code similarity tools in some cases especially for code with heavy structural changes. 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. The evaluation of ranked results provides a guideline for tool selections when one wants to retrieve only the highly similar results to the code query.

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.

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, Berlin, pp 195–195

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591

Berghel HL, Sallach DL (1984) Measurements of program similarity in identical task environments. ACM SIGPLAN Not 19(8):65

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

Box GE, Hunter JS, Hunter WG (1978) Statistics for experimenters. Wiley, New Jersey

Breuer PT, Bowen JP (1994) Decompilation: the enumeration of types and grammars. ACM Trans Program Lang Syst 16(5):1613–1647

Brixtel R, Fontaine M, Lesner B, Bazin C, Robbes R (2010) Language-independent clone detection applied to plagiarism detection. In: Proceedings of the 10th working conference on source code analysis and manipulation (SCAM '10), pp 77–86

Bruntink M, van Deursen A, van Engelen R, Tourwe T (2005) On the use of clone detection for identifying crosscutting concern code. IEEE Trans Softw Eng 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. Softw Prac Exp 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. Empir Softw Eng 19:1040–1074

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 (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. Trans Inf Theory 51(4):1523–1545

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

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

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. Secur Priv 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. Trans Educ 51(2):195–200

Cosma G, Joy M (2012) An approach to source-code plagiarism detection and investigation using latent semantic analysis. Trans Comput 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

Dangel A, Pelisse R (2011) Pmd's copy/paste detector (cpd). http://pmd.sourceforge.net/pmd-4.3.0/cpd.html, accessed date: 28 May 2017

Daniela C, Navrat P, Kovacova B, Humay P (2012) The issue of (software) plagiarism: a student view. Trans Educ 55(1):22–28

Davies J, German DM, Godfrey MW, Hindle A (2013) Software bertillonage: determining the provenance of software development artifacts. Empir Softw Eng 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

Donaldson J, Lancaster A, Sposato P (1981) A plagiarism detection system. SIGCSE '81

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 (2013) A source code similarity system for plagiarism detection. Comput J 56(1):70–86

Faidhi J, Robinson S (1987) An empirical approach for detecting program similarity and plagiarism within a university programming environment. Comput Educ 11(1):11–19

Fisher R (1935) The design of experiments. Oliver and boyd

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 date: 14 February 2016

Fowler M (2013) Catalog of refactorings. https://refactoring.com/catalog/, accessed: 28 May 2017

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 date: 14 February 2016

GuardSquare (2015) Proguard: bytecode obfuscation tool. http://proguard.sourceforge.net/, accessed date: 24 August 2015

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. Elsevier North-Holland, Inc, Amsterdam

Harris S (2015) Simian – similarity analyser, version 2.4. http://www.harukizaemon.com/simian/, accessed date: 14 February 2016

Hartmann B, Macdougall D, Brandt J, Klemmer SR (2010) What would other programs 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, 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. Trans Educ 42(2):129–133

Joy M, Griffiths N, Boyatt R (2005) The BOSS online submission and assessment system. Educ Res Comput 5(3):2–28

Kamiya T, Kusumoto S, Inoue K (2002) CCFInder: a multilinguistic token-based code clone detection system for large scale source code. Trans Softw Eng 28(7):654–670

Kapser C (2006) Godfrey m: 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

Kapser CJ, Godfrey MW (2008) Cloning considered harmful" considered harmful: patterns of cloning in software. Empir Softw Eng 13(6):645–692

Ke Y, Stolee KT, Goues CL, Brun Y (2015) Repairing programs with semantic code search. In: Proceedings of the 30th international conference on automated software engineering (ASE '15), pp 295–306

Keivanloo I, Rilling J, Zou Y (2014) Spotting working code examples. In: Proceedings of the 36th international conference on software engineering (ICSE '14), pp 664–675

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, Berlin

Li Z, Lu S, Myagmar S, Zhou Y (2006) CP-Miner: finding copy-paste and related bugs in large-scale software code. Trans Softw Eng 32(3):176–192

Hi Lim, Park H, Choi S, Han T (2009) A method for detecting the theft of Java programs through analysis of the control flow information. Inf Softw Technol 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 date: 14 February 2016

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 to information retrieval, vol 21. Cambridge University Press, Cambridge

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. J Softw Evolution Process 25:261–283

Pavlov I (2016) 7-Zip. http://www.7-zip.org, accessed: 14 February 2016

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. J Mach Learn Res 12(Oct):2825–2830

Pike R, Loki (2002) The sherlock plagiarism detector. http://www.cs.usyd.edu.au/~scilect/sherlock/, accessed date: 14 February 2016

Poulter G (2012) Python ngram 3.3. https://pythonhosted.org/ngram/, accessed date: 14 February 2016

Prechelt L, Malpohl G, Philippsen M (2002) Finding plagiarisms among a set of programs with JPlag. J Universal Comput Sci 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 date: 14 February 2016

Ragkhitwetsagul C, Krinke J (2017a) The study's website: comparison of code similarity analysers. http://crest.cs.ucl.ac.uk/resources/cloplag/, accessed date: 20 June 2017

Ragkhitwetsagul C, Krinke J (2017b) 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. Sci Comput Program 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[TM] family of source code obfuscators. http://www.semdesigns.com/products/obfuscators/, accessed: 14 February 2016

Smucker MD, Allan J, Carterette B (2007) A comparison of statistical significance tests for information retrieval evaluation. In: Proceedings of the 16th conference on information and knowledge management (CIKM '07), p 623

Strobel M (2016) Procyon / java decompiler. https://bitbucket.org/mstrobel/procyon/wiki/java%20decompiler accessed date: 14 February 2016

Stunnix (2016) Http://stunnix.com, accessed date: 14 February 2016

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

Svajlenko J, Roy CK (2016) Bigcloneeval: a clone detection tool evaluation framework with BigCloneBench. In: Proceedings of the 32nd international conference on software maintenance and evolution (ICSME '16), pp 596–600

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

Thomas SW, Hemmati H, Hassan AE, Blostein D (2014) Static test case prioritization using topic models. Empir Softw Eng 19(1):182–212

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 date: 14 February 2016

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 date: 14 February 2016

Vargha A, Delaney HD (2000) A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. J Educ Behav Stat 25:101–132. (2 (Summer, 2000))

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. Comput J 33(2):140–146

Wilcoxon F (1945) Individual comparisons by ranking methods. Biom Bull 1(6):80

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

Yang D, Martins P, Saini V, Lopes C (2017) Stack overflow in github: any snippets there? In: Proceedings of the international conference on mining software repositories (MSR '17)

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
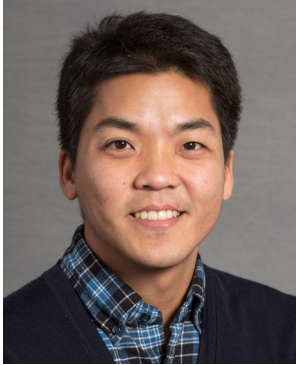
Chaiyong Ragkhitwetsagul is working toward the PhD degree in computer science at University College London, where he is part of the Centre for Research on Evolution, Search, and Testing (CREST) and SoftwareSystems Engineering (SSE) Group. He is supervised by Jens Krinke and David Clark. His research interests include code search, code clone detection, software plagiarism, and mining software repository.

Jens Krinke is Senior Lecturer in the Software Systems Engineering Group at the University College London, where he is Director of CREST, the Centre for Research on Evolution, Search, and Testing. His main focus is software analysis for software engineering purposes. His current research interests include software similarity, modern code review, and mutation testing. He is well known for his work on program slicing and clone detection.

David Clark is a Reader in Program Analysis in the Department of Computer Science at UCL and a founding member of the Theory Tendency of the department's Software Systems Engineering Research Group. He researches program analyses that support information theoretic measures on software and their applications to problems in software engineering. His recent research has seen a focus on malware detection and classification and on problems in software testing. The latter include test suite adequacy and diversity, the oracle problem, and the failed error propagation problem.