2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation

Similarity of Source Code in the Presence of Pervasive Modifications

Chaiyong Ragkhitwetsagul, Jens Krinke, David Clark
CREST, Department of Computer Science
University College London, UK

***Abstract*—Source code analysis to detect code cloning, code plagiarism, and code reuse suffers from the problem of pervasive code modifications, i.e. transformations that may have a global effect. We compare 30 similarity detection techniques and tools against pervasive code modifications. We evaluate the tools using two experimental scenarios for Java source code. These are (1) pervasive modifications created with tools for source code and bytecode obfuscation and (2) source code normalisation through compilation and decompilation using different decompilers. Our experimental results show that highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures. Our study strongly validates the use of compilation/decompilation as a normalisation technique. Its use reduced false classifications to zero for six of the tools. This broad, thorough study is the largest in existence and potentially an invaluable guide for future users of similarity detection in source code.**

I. Introduction

Assessing source code similarity is a fundamental activity in software engineering and it has many applications. These include clone detection, the problem of locating duplicated code fragments; plagiarism detection; software copyright infringement; and code search, in which developers search for similar implementations. While that list covers the more common applications, similarity assessment is used in many other areas, too. Examples include finding similar bug fixes [22], identifying cross-cutting concerns [6], program comprehension [35], code recommendation [23], and example extraction [37].

The assessment of source code similarity has a co-evolutionary relationship with the modifications made to the code at the point of its creation. In this paper we consider not only local transformations but in particular pervasive modifications, such as changes in layout or renaming of identifiers, changes that affect the code globally. Loosely, these are code transformations that arise in the course of code cloning, software plagiarism, and software evolution, but exclude strong obfuscation [10]. In code reuse by code cloning, which occurs through copying and pasting a fragment from one place to another, the copied code is often modified to suit the new environment [47]. Modifications include formatting changes and identifier renaming (Type I and II clones), structural changes, e.g. `if` to `case` or `while` to `for`, or insertions or deletions (Type III clones) [15]. Likewise, software plagiarisers copy source code of a program and modify it to avoid being caught [14]. Moreover, source code is modified during software evolution [40]. Therefore, most clone or plagiarism detection tools and techniques tolerate different degrees of change and still identify cloned or plagiarised fragments. However, while they usually have no problem in the presence of local or confined modifications, pervasive modifications that transform whole files or systems remain a challenge [46].

This work is motivated by the question: "When source code is pervasively modified, which similarity detection techniques or tools get the most accurate results?" To answer this question, we provide a thorough evaluation of the performance of the current state-of-the-art similarity detection techniques on pervasively modified code. The study presented in this paper is the largest extant study on source code similarity and covers the widest range of techniques and tools. Previous studies, e.g. on the accuracy of clone detection tools [5], [47], [54] and of plagiarism detection tools [21], were mainly focused on a single technique or tool, or on a single domain.

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

We have selected as many techniques for source code similarity measurement as possible, 30 in all, covering techniques specifically designed for clone and plagiarism detection, plus the normalised compression distance, string matching, and information retrieval. In general, the selected tools require the optimisation of their parameters as these can affect the tools' execution behaviours and consequently their results. A previous study [57] has explored the optimisation of parameters only for a small set of clone detectors. Therefore, we have explored the range of configurations for each tool, studied their impact, and discovered the configurations optimal for each data set used in our experiments.

Clone and plagiarism detection use intermediate representations like token streams or abstract syntax trees or other transformations like pretty printing or comment removal to achieve a normalised representation [47]. We integrated compilation and decompilation as a normalisation pre-process step for similarity detection and evaluated its effectiveness.

This paper makes the following primary contributions:

1. A broad, thorough study of the performance of similarity tools and techniques: We compare a large range of 30 similarity detection techniques and tools using two experimental scenarios for Java source code in order to measure the techniques' performances and observe their behaviours. The

978-1-5090-3848-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SCAM.2016.13

IEEE computer society

results show that highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures.

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

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

II. Empirical Study

Our empirical study consisted of two different experiment scenarios. The first scenario was on the products of the two obfuscation tools and to search for optimised configurations of the 30 similarity analysers. The second scenario examined the effectiveness of compilation/decompilation as a preprocessing normalisation strategy.

The 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?

RQ2 (Optimal configurations): What are the best parameter settings and similarity thresholds for the techniques?

RQ3 (Normalisation by decompilation): Does use of compilation followed by decompilation as a pre-processing normalisation method improve detection results?

A. Experimental framework

The general framework of our study as shown in Figure 1 consists of 5 main steps. In Step 1, we collect test data consisting of Java source code files. Next, the source files are transformed by applying pervasive modifications at source and bytecode level. In the third step, all original and transformed source files are normalised. A simple form of normalisation is pretty printing the source files which is used in similarity or clone detection [45]. We also use decompilation. In Step 4, the similarity detection tools are executed pairwise against the set of all normalised files, producing similarity reports for every pair. In the last step, the similarity reports are analysed.

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

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

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

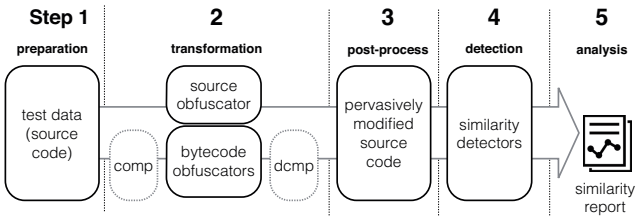

Fig. 1. The experimental framework

B. Tools and Techniques

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

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 [49] is an Eclipse plugin for source-level obfuscation. The tool makes 5 different transformations to Java source code including 1) renaming of variables, fields, and methods, 2) changing assignment, increment, and decrement operations to normal form, 3) inserting additional assignment, increment, and decrement operations when possible, 4) changing `while` to `for` and the other way around, and 5) changing `if` to its short form. Artifice cannot be automated and has to be run manually because it is an Eclipse plugin. ProGuard [44] is a well known open-source bytecode obfuscator. It is a versatile tool containing several functions including shrinking Java class files, optimisation, obfuscation, and pre-verification. ProGuard obfuscates Java bytecode by renaming classes, fields, and variables with short and meaningless ones. It also performs package hierarchy flattening, class repackaging, and modifying class and package access permissions.

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. [34] use GCC/G++ with different optimisation options to generate 10 different binary versions of the same program. However, if the desired final product is source code, a decompiler is also required in the process in order to transform the bytecode back to its source form.

Decompilation is a method for reversing the process of program compilation. Given a low-level language program such as an executable file, a decompiler generates a high-level language counterpart that resembles the (original) source code. This has several applications including recovery of lost source code, migrating a system to another platform, upgrading an old program into a newer programming language, restructuring poorly-written code, finding bugs or malicious code in binary programs, and program validation [8]. An example of using the decompiler to reuse code is a well-known lawsuit between Oracle and Google [39]. It seems that Google decompiled a Java library to obtain the source code of its APIs and then partially reused them in their Android operating system.

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

We selected two open source decompilers: Krakatau and Procyon. Krakatau [30] is an open-source tool set comprising a decompiler, a classfile dissembler, and an assembler. Procyon [42] is also a Java open-source decompiler. It has advantages over other decompilers for declaration of `enum`, `String`, `switch` statements, anonymous and named local classes, annotations, and method references. They are used in both the transformation (obfuscation) and normalisation post-process steps (Steps 2 and 3) of the framework.

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

3) Plagiarism Detectors: The selected plagiarism detectors include JPlag, Sherlock, Sim, and Plaggie. JPlag [41] and Sim [19] are token-based tools which come in versions for text (jplag-text and simtext) and Java (jplag-java and simjava), while Sherlock [50] relies on digital signatures (a number created from a series of bits converted from the source code text). Plaggie's detection [2] 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, $\mathrm{sim}(x, y)$, for a given file pair x, y.

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* [57] 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 sum of n clone fragment lines found in x (overlaps are handled) to the number of lines in x and vice versa.

$$\mathrm{sim}_{\mathrm{GCF}}(x, y) = \frac{\sum_{i=1}^{n} |\mathit{frag}_i(x)|}{|x|} \tag{2}$$

Using this method, we included five state-of-the-art clone detectors: CCFinderX, NICAD, Simian, iClones, and Deckard. CCFinderX (ccfx) [27] is a token-based clone detector detecting similarity using suffix trees. NICAD [45] is a clone detection tool embedding TXL for pretty-printing, and compares source code using string similarity. Simian [51] 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 [20] performs token-based incremental clone detection over several revisions of a program. Deckard [25] 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.

5) Compression tools: Normalised compression distance (NCD) is a distance metric between two documents based on compression [9]. It is an approximation of the normalised information distance which is in turn based on the concept of Kolmogorov complexity [32]. The NCD between two documents can be computed by

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

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 the CompLearn suite [12] 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 7Zip [1] 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

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

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

6) Other Techniques: We expanded our study with other techniques for measuring similarity including a range of libraries that measure textual similarity: difflib [16] compares text sequences using Gestalt pattern matching, NGram [38] compares text sequences via fuzzy search using n-grams, fuzzywuzzy [18] uses fuzzy string matching, jellyfish [24] does approximate and phonetic matching of strings, and cosine similarity from scikit-learn [52] which is a machine learning

TABLE I
TOOLS WITH THEIR SIMILARITY MEASURES

Tool/Technique	Similarity calculation
Clone Det.	
ccfx	tokens and suffix tree matching
deckard	characteristic vectors of AST optimised by LSH
iclones	tokens and generalised suffix tree
nicad	TXL and string comparison (LCS)
simian	line-based string comparison
Plagiarism Det.	
jplag-java	tokens, Karp Rabin matching, Greedy String Tiling
jplag-text	tokens, Karp Rabin matching, Greedy String Tiling
plaggie	N/A (not disclosed)
sherlock	digital signatures
simjava	tokens and string alignment
simtext	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	*ncd* tool with bzlib & zlib
Others	
bsdiff	Equation (5)
diff	Equation (5)
py-difflib	Gestalt pattern matching
py-fuzzywuzzy	fuzzy string matching
py-jellyfish	approximate and phonetic matching of strings
py-ngram	fuzzy search based using n-gram
py-sklearn	cosine similarity from machine learning library

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

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

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 I.

III. EXPERIMENT SCENARIOS

To answer the research questions, two experiment scenarios were designed and studied following the framework presented in Figure 1. The experiments were conducted on a virtual machine with 2.67 GHz CPU (dual cores) and 2 GB RAM running Scientific Linux release 6.6 (Carbon). The details of each scenario are explained below.

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.

1) Preparation, Transformation, and Normalisation: This section follows Steps 1 and 2 in the framework. The original data consists of 5 Java classes: *InfixConverter*, *SqrtAlgorithm*, *Hanoi*, *EightQueens*, and *MagicSquare*. All of them are short Java programs with less than 200 LOC and illustrate issues that are usually discussed in basic programming classes. The process of test data preparation and transformation is illustrated in Figure 3. First, we selected each original source code file and obfuscated it using Artifice. This produced the first type of obfuscation: source-level obfuscation (No. 1). An example of a method before and after source-level obfuscation by Artifice are displayed on the left side of Figure 2 (formatting has been adjusted due to space limits).

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

All four bytecode files were then decompiled by either Krakatau or Procyon giving back eight additional obfuscated source code files. For example, No. 1 in Figure 3 is a pervasively modified version via source code obfuscation with Artifice. No. 2 is a version which is obfuscated by Artifice, compiled, obfuscated with Proguard, and then decompiled with Krakatau. No. 3 is a version obfuscated by Artifice, compiled and then decompiled with Procyon. Using this method, we obtained 9 pervasively modified versions for each original source file, resulting in 50 files for the data set. The only post-processing step in this scenario is normalisation through pretty printing.

2) Similarity Detection: The generated data set of 50 Java code files is used for pairwise similarity detection in Step 4 of the framework in Figure 1, resulting in 2,500 pairs of source code files with their respective similarity values. We denote each pair (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. Hence, the number of reports generated by one tool equals the number of combinations of its parameter values. A tool with two parameters $p_1 \in P_1$ and $p_2 \in P_2$ has $|P1| \times |P2|$ different settings. For example, if ccfx has two parameters $b \in \{10, 20, 30, 40, 50\}$ and $t \in \{1, 2, .., 12\}$, we needed to do $5 \times 12 \times 2,500 = 150,000$ pairwise comparisons.

3) Analysing the Similarity Reports: In Step 5 of the framework, the results of the pairwise similarity detection are analysed. The 2,500 pairwise comparisons result in 2,500 (x, y, s) entries. As in Equation (1), all pairs x, y are considered to be similar when the reported similarity s 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.

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

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

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

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

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

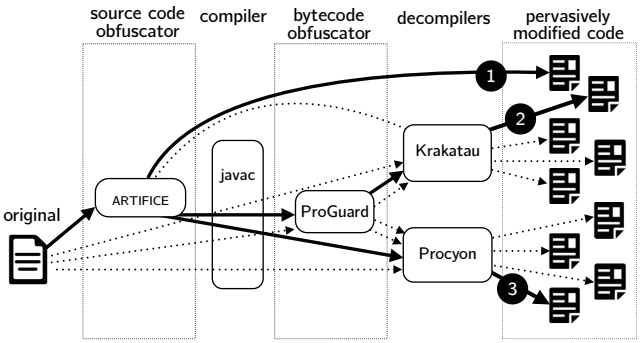

Fig. 3. Test data generation process

The ground truth of the generated data set contains 500 positives and 2,000 negatives. The positive pairs are the pairs of files generated from the same original code. For example, all pairs that are the derivatives of `InfConv.java` must be reported as similar. The other 2,000 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[1].

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

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

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

Using the F-score we can search for the best threshold T under which each tool has its optimal performance with the highest F-score. For example in Figure 4, after varying the threshold from 0 to 100, ncd-bzlib has the best threshold $T = 31$ with the highest F-score of 0.8282. 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".

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

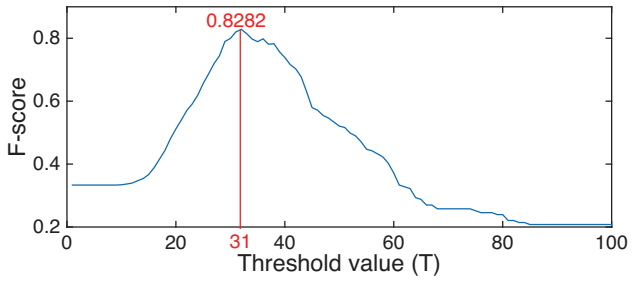

Fig. 4. The graph shows the F-score and the threshold values of ncd-bzlib. The tool reaches the highest F-score when the threshold equals 31.

Scenario 2 (Decompilation)

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

Likewise, changes made by bytecode obfuscators may also be normalised by decompilers. Suppose a Java program P is obfuscated 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, the data set is based on the same set of 50 source code files generated in Scenario 1. However, we added normalisation through decompilation to the post-processing

TABLE II
TOOL PERFORMANCE COMPARISON ON THE GENERATED DATA SET IN TERM OF ACCURACY, PRECISION, RECALL, AND F-SCORE, PRESENTED WITH THEIR BEST SETTINGS AND THRESHOLD VALUES

Tool	Settings	*T*	FP	FN	Accuracy	Precision	Recall	AUC	Prec@n	F-score
ccfx	b=20,t=1	4	42	48	0.9640	0.9145	0.9040	0.9468	0.9040	0.9095
simjava	r=22	5	64	44	0.9568	0.8769	0.9120	0.9490	0.8840	0.8941
jplag-text	t=8	2	96	52	0.9408	0.8235	0.8960	0.9453	0.8440	0.8582
py-difflib	noautojunk	35	49	103	0.9392	0.8901	0.7940	0.9147	0.8080	0.8393
7zncd-BZip2	mx=1	39	44	114	0.9368	0.8977	0.7720	0.9419	0.8180	0.8301
ncd-bzlib		31	66	100	0.9336	0.8584	0.8000	0.9482	0.8200	0.8282
jplag-java	t=3	43	142	68	0.9160	0.7526	0.8640	0.9667	0.7860	0.8045
py-sklearn		33	280	98	0.8488	0.5894	0.8040	0.9146	0.6200	0.6802

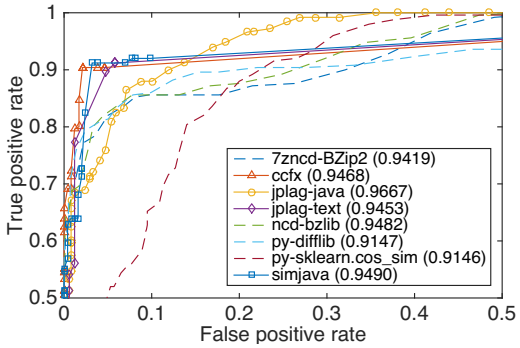

Fig. 5. The ROC curves of the 8 selected tools (zoomed in) and their respective area under the curve (AUC).

(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.

IV. RESULTS

RQ1: Performance Comparison

From Tables II and III, we can see that the tools' performance vary over the same data set. Due to the page limit, we present detailed results for only eight tools from the total set of 30 tools here (Table II): 7zncd-BZip2, ccfx, jplag-java, jplag-text, ncd-bzlib, py-difflib, py-sklearn, and simjava. We selected these 8 tools to cover every category of code similarity detection (clone detectors, plagiarism detectors, compression tools, and other tools) and to cover different similarity detection techniques. Table III gives only the F-score and the complete results of all the tools can be found from the study website[2], including the complete generated data set.

In terms of accuracy and F-score, the token-based clone detector ccfx is the winner with the highest F-score (0.9095) followed by simjava (0.8941), simian (0.8719), deckard (0.8595), and jplag-text (0.8582) respectively. In general, the best clone and plagiarism detectors outperform compression techniques and other methods. However, compression techniques and other methods outperform the worst clone and plagiarism detectors.

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

Interestingly, while we include many NCD tools with different compression algorithms in our study, the complete results in Table III show that they generate comparable results. The three bzip2-based NCD implementations, 7zncd-BZip2, ncd-bzlib, bzip2ncd, and xzncd only slightly outperform other compressors like gzip or LZMA. So the actual compression method may not have a strong effect in this context.

From the overall performance found from varying the similarity threshold from 0 to 100, we drew the receiver operating characteristic (ROC) curves for the selected eight tools, calculated the area under the curve (AUC), and compared them. The closer the value is to one, the better the tool's performance. We can see from Figure 5 that jplag-java is the winner in this analysis with the highest AUC (0.9667), followed by simjava (0.9490), ncd-bzlib (0.9482), and ccfx (0.9468). The two other methods, py-sklearn and py-difflib, are the last with an AUC of 0.9146 and 0.9147.

We show accuracy, true and false positives, precision and recall of the tools in Table II. The best tool with respect to false positives, accuracy, precision, and F-score is ccfx, with respect to false negatives and recall is simjava, and with respect to AUC the best tool is jplag-java.

Additionally, we have repeated the process of finding the optimal threshold each time we changed to a new data set. The configuration problem for clone detection tools including setting thresholds has been mentioned by several studies as one of the threats to validity [56]. There has also been an initiative to avoid using thresholds completely for clone detection [28].

To avoid this problem of threshold sensitivity we employ a measurement mainly used in information retrieval called "precision at *n* (or precision at *k*)" which shows how well a tool retrieves relevant results within top-*n* ranked items [36].

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

To achieve this, we sort the retrieved pairs by measured similarity and consider only the top n pairs with highest similarities, where n is the number of pairs in the ground truth, i.e. $n = 500$. The results in Table II show that evaluating the tools using precision at *n* gives almost the same F-scores, having ccfx as the best tool. Only the ranking of py-difflib, 7zncd-BZip2, and ncd-bzlib are reversed but with small differences in the prec@*n* scores. Hence, this precision at *n* error measure can

TABLE III
BEST CONFIGURATIONS OF EVERY TOOL AND TECHNIQUE OBTAINED FROM THE GENERATED DATA SET IN SCENARIO 1

Technique	Settings	*T*	F-score	Rank
Clone det.				
ccfx	b=20,21,24,t=1..7	4	0.9095	1
	b=22,23,t=7	2		
deckard	MINTOKEN=30	5	0.8595	4
	STRIDE=2			
	SIMILARITY=0.95			
iclones	minblock=10	0	0.6033	28
	minclone=50			
nicad	abstractexpressions	0	0.7080	24
simian	threshold=5,ignoreidentifiers	0	0.8719	3
Plagiarism det.				
jplag-java	t=3	43	0.8045	21
jplag-text	t=8	2	0.8582	5
plaggie	M=7	18	0.8210	12
sherlock	N=6,Z=3	1	0.8284	8
simjava	r=22	5	0.8941	2
simtext	r=4	17	0.5622	30
Compression				
7zncd-BZip2	mx=1,3,5	39	0.8301	7
7zncd-LZMA	mx=7,9	33	0.8160	16
7zncd-LZMA2	mx=7,9	34	0.8189	13
7zncd-Deflate	mx=9	30	0.8157	17
7zncd-Deflate64	mx=9	30	0.8142	19
7zncd-PPMd	mx=9	35	0.8078	20
bzip2ncd	C=1..9	32	0.8219	11
gzipncd	C=9	25	0.8153	18
icd	ma=Deflate, Deflate64,mx=9	37	0.7404	23
ncd-zlib	N/A	28	0.8163	15
ncd-bzlib	N/A	31	0.8282	9
xz-ncd	-e	31	0.8228	10
Others				
bsdiff	N/A	71	0.5797	29
diff	N/A	7	0.6996	25
py-difflib	SM_noautojunk	35	0.8393	6
py-fuzzywuzzy	token_set_ratio	80	0.8167	14
py-jellyfish	jaro_distance	76	0.6169	27
py-ngram	N/A	43	0.7925	22
py-sklearn	N/A	33	0.6802	26

be chosen instead of the F-score in scenarios where searching for all possible threshold values is expensive.

RQ2: Optimal Configurations

We thoroughly analysed various configurations of every tool and found that some specific settings are sensitive to pervasively modified code while others are not. The complete list of the best configurations of every tool from Scenario 1 can be found in Table III. 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 0.5591 due to a very high number of false negatives. Interestingly, a previous study on agreement of clone detectors [57] observed the same difference between default and optimal configurations and the reported optimal settings are similar to the ones we found in Table III.

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

TABLE IV
CCFX'S PARAMETER SETTINGS FOR THE HIGHEST PRECISION AND RECALL

Error measure	Value	ccfx's parameters	
		b	*t*
Precision	0.964	20,21,22,24	1..7
		23	7
Recall	1.000	10	6
		17	1..8
		18..21	12
		22..25,30,40	10..12
		45,50	1..12

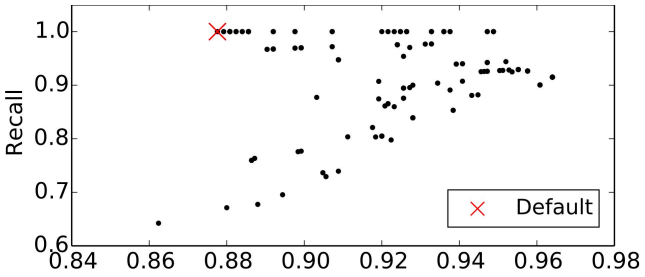

Fig. 6. Trade off between precision and recall of 217 ccfx parameter settings. The default settings provides low precision and recall.

clone fragments in the unit of tokens, and t, the minimum number of kinds of tokens in clone fragments. From Figure 6, we can see that the default settings of ccfx, b=50, t=12, (denoted with a $\times$ symbol) provides an optimal recall but low precision. We observed that one cannot tune ccfx to obtain the highest precision without sacrificing recall. The best settings for precision and recall of ccfx are described in Table IV.

Furthermore, we analysed the landscape of all 216 ccfx's parameter settings (excluding the default) in terms of F-score as depicted in Figure 7. Visually, we can distinguish a region that is the sweet spot for ccfx's parameters settings against pervasive modifications from the rest. The region covers the b value from 18 to 25, and t value from 1 to 7. The region provides the F-scores ranging from 0.8898 up to 0.9095.

RQ3: Normalisation by Decompilation

The results after adding compilation and decompilation for normalisation to the post-processing step before performing similarity detection is shown in Table V. The table shows the results of using both Krakatau and Procyon as decompiler compared to the results from Scenario 1. From the table, we can see that normalisation by compilation/decompilation has a strong effect on the number of false results reported by the tools. For Krakatau, every tool has the amount of false positives and negatives greatly reduced. In particular the leading tools from Scenario 1, ccfx and simjava even no longer report any false result (together with deckard, jplag-java, plaggie, and sherlock). All compression or other techniques still report some false results.

To confirm this, we carefully investigated the source code after normalisation and found that decompiled files created by Krakatau are very similar despite the applied obfuscation. As depicted in Figure 2 on the right side, the two code

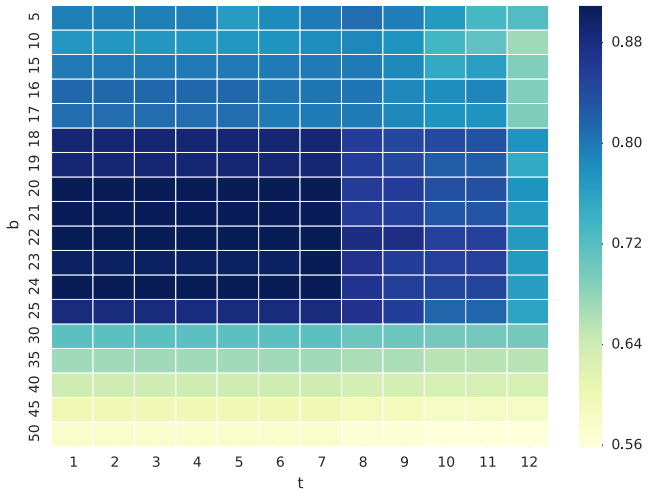

Fig. 7. F-scores of various ccfx's b and t parameter values

TABLE V
TOOL PERFORMANCE COMPARISON AFTER COMPILED/DECOMPILED USING THE DATA SET'S OPTIMAL CONFIGURATIONS.

Tool/Technique	generated		Krakatau			Procyon		
	FP	FN	FP	FN	F-score	FP	FN	F-score
ccfx	42	48	0	0	1.0000	0	4	0.9960
deckard	44	90	0	0	1.0000	0	16	0.9837
iclones	0	284	0	56	0.9407	0	166	0.8010
nicad	0	226	40	24	0.9370	0	72	0.9224
simian	2	112	2	0	0.9980	14	14	0.9720
jplag-java	142	68	0	0	1.0000	24	20	0.9562
jplag-text	96	52	16	0	0.9843	28	8	0.9647
plaggie	83	94	0	0	1.0000	0	40	0.9583
sherlock	60	104	0	0	1.0000	16	0	0.9843
simjava	64	44	0	0	1.0000	8	0	0.9921
simtext	170	238	0	24	0.9754	58	0	0.9452
7zncd-BZip2	44	114	40	12	0.9494	106	40	0.8630
7zncd-LZMA	105	83	47	5	0.9501	56	64	0.8790
7zncd-LZMA2	74	102	47	4	0.9511	56	63	0.8802
7zncd-Deflate	104	84	46	6	0.9500	52	73	0.8723
7zncd-Deflate64	103	86	46	6	0.9500	52	73	0.8723
7zncd-PPMd	108	88	49	2	0.9513	52	69	0.8769
bzip2ncd	102	80	40	16	0.9453	90	40	0.8762
gzipncd	58	116	40	8	0.9535	61	40	0.9011
icd	112	140	39	93	0.8605	60	93	0.8418
ncd-bzlib	66	100	46	14	0.9419	88	44	0.8736
ncd-zlib	67	109	50	5	0.9474	61	44	0.8968
xz-ncd	98	82	46	0	0.9560	58	56	0.8862
bsdiff	66	269	8	78	0.9075	28	149	0.7986
diff	238	103	52	65	0.8815	27	76	0.8917
py-difflib	49	103	16	73	0.9056	12	40	0.9465
py-fuzzywuzzy	68	108	0	28	0.9712	0	36	0.9627
py-jellyfish	222	178	38	146	0.7937	32	192	0.7333
py-ngram	76	122	32	56	0.9098	58	64	0.8773
py-sklearn	280	98	98	0	0.9107	50	0	0.9524

fragments become very similar after compile and decompile by Krakatau. This is because Krakatau has been designed to be robust to minor obfuscations and the transformations made by Artifice and Proguard are not very complex. Normalisation via decompilation with Procyon also improves the performance of the similarity detectors, but not as much as Krakatau. Interestingly, Procyon performs slightly better for diff, py-difflib, and py-sklearn.

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

Discussion

In summary, we have answered the three research questions after investigating the two experiment scenarios. We found that the state-of-the-art tools perform differently on pervasively modified code. Properly configured, a well known and often used clone detector, ccfx, performs best, closely followed by a plagiarism detector, simjava.

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

We analysed the search space of configurations of ccfx and found that there is a specific region of parameter settings that drive ccfx to the highest performance (F-score) against pervasive modifications. This set of parameter settings can be used as a guideline for other code similarity research. Moreover, we illustrate that one can trade off between precision and recall of ccfx by adjusting its parameters and pick the one that suits the purposes. 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.

V. THREATS TO VALIDITY

Internal validity: We carefully chose the data sets for our experiment. We created the first data set (generated) by ourselves to obtain the ground truth for positive and negative results. However, the obfuscators (Artifice and ProGuard) possibly may not represent typical pervasive modifications.

Although we have attempted to use the tools with their best parameter settings, we cannot guarantee that we have and it may be possible that the performance of some detectors is due to wrong usage instead of the techniques used in the detector.

Moreover, in this study we mainly compare performance based on standard measurements of accuracy and F-score. There might be some situations where precision or recall is preferred over another and that might produce different results.

External validity: The tools used in this study are restricted to being open-source ones or at least freely available. They 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). They might not be sufficiently representative of all available techniques and tools.

In addition, the two decompilers (Krakatau, Procyon) are only a subset of all decompilers available. So they may not be sufficiently representative of the performances of the other decompilers in the market or even other source code normalisation techniques. We chose two so that we could compare their behaviours and performances. As we are exploiting features of Java source and byte code, our findings only apply to Java code. Lastly, it may not be possible to apply decompilation to a given Java code file depending on the dependencies in the source code and the chosen decompilers.

VI. Related Work

Plagiarism is obviously a problem of serious concern in education. Similarly in industry, the copying of code or programs is copyright infringement. They both affect the originality of one's idea, his or her credibility, and also the quality of their organisation. The problem of software plagiarism has been occurring for several decades in schools and universities [13], [14] and in law, where one of the more visible cases regarding copyright infringement of software is the ongoing lawsuit between Oracle and Google [39].

To detect plagiarism or copyright infringement of source code, one has to measure similarity of two programs. Two programs can be similar at the level of purpose, algorithm, or implementation [61]. Most of software plagiarism tools and techniques focus on the level of implementation since it is most likely to be plagiarised. The process of code plagiarism involves pervasive modifications to hide the plagiarism which often includes obfuscation. The goal of code obfuscation is to make the modified code harder to understand by humans and harder to reverse engineer while preserving its semantics [10], [11], [58]. Deobfuscation attempts to reverse engineer obfuscated code [55]. Because Java bytecode is comparatively high-level and easy to decompile, obfuscation of Java bytecode has focused on preventing decompilation [3] while decompilers like Krakatoa [43], Krakatau [30] and Procyon [42] attempt to decompile even in the presence of obfuscation.

Several similarity detection tools for source code and binary code have been introduced by the research community. Many of them are based on string comparison techniques such as Longest Common Subsequence (LCS) found in NICAD [45], Plague [58], YAP [59], and CoP [34]. Many tools transform source code into an intermediate representation such as tokens and apply similarity measurement on them (Plague [58], Sherlock [26], Sim [19], YAP3 [60], JPlag [41], CCFinder [27], CP-Miner [33], iClones [20], MOSS [48] and a few more [7], [17], [53]). Structural similarity of cloned code can be discovered by using abstract syntax trees as found in CloneDR [4] and Deckard [25] or by using program dependence graphs [29], [31]. The transformation into an intermediate representation like a token stream or an abstract syntax tree can be seen as a kind of normalisation. NICAD [45] uses pretty printing as part of the normalisation process for clone detection.

Although there have been a large number of clone detectors, plagiarism detectors, and code similarity detectors invented in the research community, there exist few studies that compare and evaluate their performances. Bellon et al. [5] proposed a framework for comparing and evaluating clone detectors and six tools were chosen for the studies. Later, Roy et al. [47] 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 [54]. Hage et al. [21] compare five plagiarism detectors in term of their features and performances against 17 code modifications.

The work that is closest to ours is the empirical study of the efficiency of current detection tools against code obfuscation [49]. The authors created the Artifice source code obfuscator and measured the effects of obfuscation on clone detectors. However, the tools chosen for the study were limited to only three detectors: JPlag, CloneDigger, and Scorpio. This study showed that token-based clone detection outperformed text-, tree- and graph-based clone detection (similar to our findings).

Roy et al. [46] 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, with systematic renaming the only pervasive modification. Due to this limitation, 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 [45].

VII. Conclusions

This study of similarity detection on pervasively modified source code 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 extensive variation in performance when they are run against two 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. Moreover, through systematic investigation we determined the range of the optimal parameter settings for employing the tool ccfx against pervasive modifications and the effects of the settings on its precision and recall.

Finally, we confirmed that compilation and decompilation can be used as an effective normalisation method that greatly improves similarity detection on Java source code, leading to 6 clone and plagiarism tools not reporting any false classifications on our generated data set.

REFERENCES

[1] 7-zip. http://www.7-zip.org, Accessed: 2015-08-27.

[2] A. Ahtiainen, S. Surakka, and M. Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *6th Baltic Sea Conference on Computing Education Research*, 2006.

[3] M. Batchelder and L. Hendren. Obfuscating Java: The most pain for the least gain. In *Compiler Construction*, pages 96–110, 2007.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM'98*, pages 368–377, 1998.

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

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

[7] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, 37(2):151–175, Feb. 2007.

[8] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[9] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[10] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science University of Auckland, 1997.

[11] C. S. Collberg, C. Thomborson, and S. Member. Watermarking, tamper-proofing, and obfuscation. *Computer*, 28(8):735–746, 2002.

[12] Complearn. http://complearn.org/index.html, Accessed: 2015-08-24.

[13] G. Cosma and M. Joy. Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, 51(2):195–200, May 2008.

[14] C. Daniela, P. Navrat, B. Kovacova, and P. Humay. The issue of (software) plagiarism: A student view. *IEEE Transactions on Education*, 55(1):22–28, Feb. 2012.

[15] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3-4):219–36, 1995.

[16] difflib – helpers for computing deltas. http://docs.python.org/2/library/difflib.html, Accessed: 2015-08-24.

[17] Z. Duric and D. Gasevic. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, Mar. 2012.

[18] Fuzzy string matching like a boss. https://github.com/seatgeek/fuzzywuzzy, Accessed: 2015-08-24.

[19] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. In *30th SIGCSE technical symposium on Computer science education – SIGCSE '99*, pages 266–270, 1999.

[20] N. Göde and R. Koschke. Incremental clone detection. In *CSMR'09*, pages 219–228, 2009.

[21] J. Hage, P. Rademaker, and N. van Vugt. A comparison of plagiarism detection tools. Technical Report UU-CS-2010-015, Department of Information and Computing Sciences, Utrecht University, 2010.

[22] B. Hartmann, D. Macdougall, J. Brandt, and S. R. Klemmer. What would other programmers do? suggesting solutions to error messages. In *ACM Conference on Human Factors in Computing Systems*, 2010.

[23] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, 2005.

[24] A python library for doing approximate and phonetic matching of strings. https://github.com/jamesturk/jellyfish, Accessed: 2015-08-24.

[25] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, 2007.

[26] M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3), 2005.

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

[28] I. Keivanloo, F. Zhang, and Y. Zou. Threshold-free code clone detection for a large-scale heterogeneous java repository. In *SANER*, 2015.

[29] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS'01*, pages 40–56, 2001.

[30] Java decompiler, assembler, and disassembler. https://github.com/Storyyeller/Krakatau, Accessed: 2015-08-27.

[31] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conference on Reverse Engineering*, pages 301–309, 2001.

[32] M. Li and P. M. B. Vitâanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 2008.

[33] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[34] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE'14*, pages 389–400, 2014.

[35] J. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, 2001.

[36] C. D. Manning, P. Raghavan, and H. Schutze. *An Introduction toInformation Retrieval*, volume 21. Cambridge University Press, 2009.

[37] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus. How can I use this method? In *ICSE*, 2015.

[38] Ngram 3.3. https://pythonhosted.org/ngram/, Accessed: 2015-08-24.

[39] Oracle America, Inc. v. Google Inc., No. 3:2010cv03561 - Document 642 (N.D. Cal. 2011). http://law.justia.com/cases/federal/district-courts/FSupp2/259/597/2362960/, 2011. Online; access 23-Apr-2015.

[40] J. R. Pate, R. Tairas, and N. A. Kraft. Clone evolution: A systematic review. *Journal of software: Evolution and Process*, 25:261–283, 2013.

[41] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[42] Procyon / java decompiler. https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler, Accessed: 2015-08-27.

[43] T. a. Proebsting and S. a. Watterson. Krakatoa: Decompilation in Java (does bytecode reveal source?). In *Usenix*, pages 185–198, 1997.

[44] ProGuard: bytecode obfuscation tool. http://proguard.sourceforge.net, Accessed: 2015-08-24.

[45] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC'08*, pages 172–181, 2008.

[46] C. K. Roy and J. R. Cordy. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166. IEEE, 2009.

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

[48] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD'03*, 2003.

[49] S. Schulze and D. Meyer. On the robustness of clone detection to code obfuscation. In *IWSC'13*, pages 62–68, 2013.

[50] The sherlock plagiarism detector. http://sydney.edu.au/engineering/it/~scilect/sherlock/, Accessed: 2015-08-27.

[51] Simian - similarity analyser. http://www.harukizaemon.com/simian/, Accessed: 2015-08-27.

[52] Machine learning in python. http://scikit-learn.org/stable/, Accessed: 2015-08-24.

[53] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *IWSC'09*, 2009.

[54] J. Svajlenko and C. K. Roy. Evaluating modern clone detection tools. In *ICSME'14*, pages 321–330, 2014.

[55] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *WCRE*, 2005.

[56] C. W. C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *International Conference on Dependable Systems and Networks*, 2001.

[57] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *FSE'13*, pages 455–465, 2013.

[58] G. Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2):140–146, Feb. 1990.

[59] M. J. Wise. Detection of similarities in student programs. In *SIGCSE Technical Symposium on Computer Science Education*, 1992.

[60] M. J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *27th SIGCSE Technical Symposium on Computer Science Education*, pages 130–134, 1996.

[61] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.