Expert Systems With Applications 97 (2018) 405–420

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

A detection framework for semantic code clones and obfuscated code

Abdullah Sheneamer[a,b,*], Swarup Roy[c,d], Jugal Kalita[b]

[a] Faculty of Computer Science & Information Systems, Jazan University, Jazan 45142, Saudi Arabia
[b] College of Engineering & Applied Science, University of Colorado, Colorado Springs, CO 80918, USA
[c] Department of Computer Applications, Sikkim University, Sikkim, Gangtok 737102, India
[d] Department of Information Technology, North-Eastern Hill University, Shillong 793022, India

ARTICLE INFO

Article history:
Received 17 May 2017
Revised 21 December 2017
Accepted 22 December 2017
Available online 29 December 2017

Keywords:
Code obfuscation
Semantic code clones
Machine learning
Bytecode dependency graph
Program dependency graph

ABSTRACT

Code obfuscation is a staple tool in malware creation where code fragments are altered substantially to make them appear different from the original, while keeping the semantics unaffected. A majority of the obfuscated code detection methods use program structure as a signature for detection of unknown codes. They usually ignore the most important feature, which is the semantics of the code, to match two code fragments or programs for obfuscation. Obfuscated code detection is a special case of the semantic code clone detection task. We propose a detection framework for detecting both code obfuscation and clone using machine learning. We use features extracted from Java bytecode dependency graphs (BDG), program dependency graphs (PDG) and abstract syntax trees (AST). BDGs and PDGs are two representations of the semantics or meaning of a Java program. ASTs capture the structural aspects of a program. We use several publicly available code clone and obfuscated code datasets to validate the effectiveness of our framework. We use different assessment parameters to evaluate the detection quality of our proposed model. Experimental results are excellent when compared with contemporary obfuscated code and code clone detectors. Interestingly, we achieve 100% success in detecting obfuscated code based on recall, precision, and F1-Score. When we compare our method with other methods for all of obfuscations types, viz, contraction, expansion, loop transformation and renaming, our model appears to be the winner. In case of clone detection our model achieve very high detection accuracy in comparison to other similar detectors.

© 2017 Elsevier Ltd. All rights reserved.

1. Introduction

Code obfuscation is a technique that alters the original content of the code in order to sow confusion. Malware creators use obfuscation to camouflage existing malicious code and make the task of signature based malware detection tools challenging. Automated code obfuscation tools make malware development a child's play even for non-professional attackers. Traditional malware detection tools based on a priori known signatures become ineffective due to the morphing of earlier known malicious code. As results, computer systems become more prone to zero-day-attacks. The positive use of code obfuscation is equally important for protecting intellectual property rights of proprietary software. Code obfuscation prevents attackers from malicious reverse engineering of sensitive parts of a software project and helps prevent software piracy (Collberg & Thomborson, 2002). However, we concentrate only on the negative aspect of the problem, i.e., assuming that code obfuscation is malicious and detecting it is necessary. The common way to perform camouflaging is to change the syntactic structure of the code, keeping the semantics and functionality of the original malware invariant. Obfuscation techniques transform a code block in two different ways, using metamorphism and polymorphism (Christodorescu, Jha, Seshia, Song, & Bryant, 2005). Metamorphism obfuscates the entire code by inserting certain dead code, and by performing code substitution and code transposition. On the other hand, polymorphism uses transformations to obfuscate loops in the code. Existing malware detectors treat malware code as a sequence of bytes and extract a signature to classify it by matching with a database of known malware signatures. Syntactic or structural signatures are weak and ineffective in detecting camouflaged code and are overlooked easily by signature based malware detectors. Effective anti-malware software based on semantic structure of the code is a current need to mitigate the issue of ever-evolving variants of known malware.

* Corresponding author at: College of Engineering & Applied Science, University of Colorado, Colorado Springs, CO 80918, USA and Faculty of Computer Science & Information Systems, Jazan University, Jazan 45142, Saudi Arabia

E-mail addresses: asheneam@uccs.edu (A. Sheneamer), sroy01@cus.ac.in, swarup@nehu.ac.in (S. Roy), jkalita@uccs.edu (J. Kalita).

https://doi.org/10.1016/j.eswa.2017.12.040
0957-4174/© 2017 Elsevier Ltd. All rights reserved.

In this work, we focus only Java code and develop a machine learning framework for effective detection of code obfuscation. We model obfuscated code detection as a kind of semantic code clone detection. We use syntactic and semantic features of pairs of original and target code fragments for detection of possible obfuscated code.

We explore new features which are extracted from Java bytecode and BDGs and then combine them with features from ASTs and PDGs. The contributions of the paper are as follows.

- We propose an integrated framework for detecting Java code clones and obfuscated code using program or code features extracted from target pairs of codes.
- We use high level source code features from ASTs and PDGs of the code.
- We explore a new way of using low level features from Java bytecode and BDG to detect code clones and obfuscation. To the best of our knowledge this attempt is a first of its kind to use features from both Java bytecode and BDGs to detect semantic code clones and obfuscation using machine learning.
- We use an ensemble of state-of-the-art classification models to evaluate the effectiveness of our proposed idea.

We organize the paper as follows. Section 2 discusses the background of code clone detection and code obfuscation. Prior research in the area is discussed in Section 3. In Section 4, we propose a new machine learning framework for detection of both code clones and obfuscated code. We evaluate and compare our proposed method and report findings in Section 5. Finally, we conclude our work in Section 7.

2. Background

Code clone detection is a well-known problem in software engineering (Juergens, Deissenboeck, Hummel, & Wagner, 2009). Clones are broadly classified into two types, syntactic and semantic clones (Roy & Cordy, 2007; Roy, Cordy, & Koschke, 2009). Very often syntactic clones are further classified into Exact Clones (Type I), Renamed Clones (Type II) and Gapped Clones (Type III).

Two pieces of code are Exact Clones of each other if they are exactly the same except whitespace, blanks and comments. Renamed Clones are similar except for names of variables, types, literals and functions. Gapped Clones are the clones if they are similar, but with modifications such as added or removed statements, and the use of different identifiers, literals, types, whitespace, layouts and comments. Two pieces of code are Semantic Clones, if they are semantically similar without being syntactically similar.

Code obfuscation is a form of semantic code cloning where two malicious pieces of code may be structurally or syntactically dissimilar, but semantically behave in a similar way. Below we define code obfuscation in a more formal way. We start with what do we mean by code.

Definition 1 (Code)**.** A piece of code $\mathcal{C}$ is a sequence of statements, $S_i, i = 1, \cdots, M$, comprising of programming language specific executable statements such as loops, logical statements and arithmetic expressions:

$$\mathcal{C} = \langle S_1, \cdots S_M \rangle.$$

Before defining code obfuscation we need to define the syntactic and semantic similarities between two pieces of code.

Let $\mathcal{C}_i = \langle S_{i1}, \cdots, S_{iN_i} \rangle$, and $\mathcal{C}_j = \langle S_{j1}, \cdots, S_{jN_j} \rangle$ be two pieces of code. Assume $\mathcal{C}_i^t = trim(\mathcal{C}_i)$ where $trim(.)$ is a function that removes whitespace, blanks and comments from the code and its statements and normalizes it. Thus, whitespace that cover an entire line are removed, as well as whitespace within statements, whitespace and blanks are trimmed so that the token sequence in $\mathcal{C}_i$ and $\mathcal{C}_i^t$ are the same.

Definition 2 (Syntactic Similarity)**.** Two pieces of code $\mathcal{C}_i$ and $\mathcal{C}_j$ are structurally or syntactically similar if (i) they are both of the same length after trimming, and (ii) $\mathcal{C}_i$ and $\mathcal{C}_j$ are literally equivalent with respect to their contents.

The syntactic similarity can be represented as a discrete function $SynSim(\mathcal{C}_i, \mathcal{C}_j)$, with a binary outcome, i.e., true (1) or false (0).

$$SynSim(\mathcal{C}_i, \mathcal{C}_j) = \begin{cases} 1, & \text{if } |\mathcal{C}_i^t| = |\mathcal{C}_j^t| \text{ and } StringSim(S_{ik}^t, S_{jk}^t) > \eta, \forall k = 1 \cdots |\mathcal{C}_i^t| \\ 0, & otherwise \end{cases} \quad (1)$$

where *StringSim*(.,.) computes the literal similarity between the two given pieces of code. Two strings are exactly similar if they have the same characters in the same sequence. Two pieces of code are considered similar if $\forall k, k = 1, \cdots, |\mathcal{C}_i^t|$, all statements are syntactically similar above a threshold. The superscript t means after trimming. *StringSim*(.,.) is 1 if both the two pieces of code are exactly similar. 0 indicates they are completely different from each other. η determines the level of similarity to be considered for this relationship.

We draw a resemblance between code obfuscation and semantic cloning of code. Two pieces of code are semantic clones if they are functionally similar. Precisely defining the term semantic similarity between two pieces of code is hard. In comparison to syntactic similarity, which compares program texts and is relatively easy to do, semantic similarity is difficult to identify as it deals with the meaning or purpose of the codes, without regards to textual similarity.

The idea of semantic similarity is not easy to grasp because it requires some level of understanding the meanings of programs, whether formal or otherwise. The formal semantics of a program or a piece of code can be described in several ways, the predominant ones being denotational semantics, axiomatic semantics and operational semantics (Gunter, 1992; Winskel, 1993). Denotational semantics composes the meaning of a program or a fragment of code by composing it from the meaning (or denotation, a mathematical expression or function) of its components in a bottom-up fashion. Two pieces of code have the same meaning if their composed denotations are the same. Axiomatic semantics defines the meaning of a program or code fragment by first defining the meanings of individual commands by describing their effects on assertions about variables that represent program states, and then writing logical statements with them. In this paradigm, two pieces of code that write an algorithm slightly differently but produce the same results are considered semantically equivalent, provided their initial assertions are the same. Operational or concrete semantics does not attach mathematical meanings to components within a program or code fragment, but describes how the individual steps of a piece of code or program takes place in a computer-based system on some abstract machine. No matter which approach is used for describing formal semantics, the meaning of a code fragment or program is obtained from the meanings ascribed to the individual components. To obtain the semantics of a code fragment or program, it is initially parsed into syntactic or structural components, and for each syntactic component, its corresponding meaning is obtained, and finally the meaning of the piece of code is put together from these components, following appropriate rules. Thus, we could say two pieces of code $\mathcal{C}_i$ and $\mathcal{C}_j$ are semantically similar if

$$SemSim(\mathcal{C}_i, \mathcal{C}_j) = SemSim^*([\![\mathcal{C}_i]\!], [\![\mathcal{C}_j]\!]) > \varphi, \quad (2)$$

where $SemSim^*(.,.)$ is a formal measure of semantic similarity between the two pieces of code. $[\![C_i]\!]$ is the formal semantics of code fragment C_i computed following a formal approach. In this paper, we will not delve deeper into how $[\![C_i]\!]$, $[\![C_j]\!]$ or, $SemSim^*(.,.)$ may be computed exactly following semantic theory. In practice, we approximate the computation of $SemSim(\mathcal{C}_i, \mathcal{C}_j)$ using other means as discussed in this paper. In other words, the primary focus in this paper is discussing a way to compute an approximate computation of $SemSim^*(.,.)$ using non-formal semantic means. Assuming, we can provide a good computational approximation to $SemSim^*(.,.)$, we can proceed to define semantic clones. We call this approximation $SemSim^*(.,.)$.

Definition 3 (Semantic Clone)**.** Code $\mathcal{C}_i$ is a semantic clone of code $\mathcal{C}_j$ (or vice versa) if they are semantically similar without being syntactically similar.

Abstract definition of obfuscated code can be obtained using the notions of syntactic and semantic similarities in the following way.

Definition 4 (Code Obfuscation)**.** A code $\mathcal{C}_j$ is obfuscated version of another code $\mathcal{C}_i$ if they exhibit similar functionality although the structurally they are different from each other. It is similar to semantic clone. In other words, detection of an obfuscated code pair can be represented as a discrete function as follows.

$$Obfuscated(\mathcal{C}_i, \mathcal{C}_j) = \begin{cases} 1, & \text{if } SemSim(\mathcal{C}_i, \mathcal{C}_j) > \varphi \text{ and } SynSim(\mathcal{C}_i, \mathcal{C}_j) < \varphi; \\ 0, & \text{otherwise,} \end{cases} \quad (3)$$

where $SynSim(.,.)$ is the syntactic similarity function from Eq. (1) and $SemSim(.,.)$ is the semantic similarity function from Eq. (2), and φ is a user set threshold.

Several effective methods have been proposed for discovering obfuscated code. A brief sketch of these methods is given below.

3. Prior research

A significant amount of research has been performed in both areas of code clone and obfuscation. A number of effective tools and computational methods have been developed to address the issues of code clone detection and detecting obfuscated code independently. We discuss prior research on clone detection methods first, followed by obfuscated code detection.

Komondoor and Horwitz (2001) use, for the first time, the idea of Program Dependency Graphs (PDG) in clone detection. They use a slicing technique to find isomorphic PDG subgraphs to detect Type-I to III clones. CCFinder (Kamiya, Kusumoto, & Inoue, 2002) uses a suffix tree-matching algorithm to detect Type-I and II code clones, and it is not effective in detecting Type-III or IV clones. Deckard (Jiang, Misherghi, Su, & Glondu, 2007) computes characteristic vectors and can detect re-ordered statements and noncontiguous clones but not semantically similar clones. (Li, Lu, Myagmar, & Zhou, 2006) use a token-based technique to detect code clones and clones related to bugs in large software systems. Their system, *CP-Miner*, searches for copy-pasted code blocks using frequent subsequence mining. NiCad (Roy & Cordy, 2008) is a text-based hybrid clone detection technique, which can detect up to Type-III clones. Hummel, Juergens, Heinemann, and Conradt (2010) use a hybrid and incremental index based technique to detect clones and implement a tool called *ConQAT*. It detects only Type-I and II clones. Yuan and Guo (2011) use a count matrix to detect code clones. The method is limited to detecting only Type-I, II, and III clones. Higo, Yasushi, Nishino, and Kusumoto (2011) propose a PDG-based incremental two-way slicing approach to detect clones, called Scorpio. This approach detects non-contiguous clones while other existing incremental detection approaches cannot detect non-contiguous clones. The approach also has faster speed compared to other existing PDG based clone detection approaches. Yuan and Guo (2012) use a token-based approach called *Boreas* to detect clones. *Boreas* uses a novel counting method to obtain characteristic matrices that identify program segments effectively. Boreas is not able to detect code clones of Type-III and IV. Murakami, Hotta, Higo, Igaki, and Kusumoto (2012) develop a token-based technique called FRISC which transforms every repeated instruction into a special form and uses a suffix array algorithm to detect clones. The authors performed experiments with eight target software systems, and found that the precision with folded repeated instructions is higher than the precision without by 29.8%, but the recall decreases by 2.9%. Murakami, Hotta, Higo, Igaki, and Kusumoto (2013) develop another token-based technique, which detects Type-III clones (gapped clones) using the Smith–Waterman algorithm (Smith & Waterman, 1981), called CDSW. Because CDSW does not normalize all variables and identifiers, it cannot detect clones that have different variable names. Agrawal and Yadav (2013) extend Boreas to detect clones by using a token-based approach to match clones with one variable or a keyword and easily detect Type-I and Type-II clones; they use a textual approach to detect Type-III clones. Since Agrawal *et al.*'s approach combines two approaches, it is a hybrid approach. Hotta, Yang, Higo, and Kusumoto (2014) compare and evaluate methods for detection of coarse-grained and fine-grained unit-level clones. They use a coarse-grained detector that detects block-level clones from given source files. Their approach is much faster than a fine-grained approach, since the authors use hash values of texts of blocks. However, using a coarse-grained approach alone is not enough because it does not have more detailed information about the clones. A fine-grained approach must be used as a second stage after a coarse-grained approach. Sheneamer and Kalita (2015) propose a hybrid clone detection technique to detect only syntactic clones. SourcererCC (Saini, Sajnani, Kim, & Lopes, 2016) is a token-based syntactic and semantic clone detection method to achieve large-scale detection.

Automatic analysis of malware by detecting obfuscated code is an active research area. A number of approaches have been reported in the literature. Likarish, Jung, and Jo (2009) propose a method for JavaScript obfuscated code detection. Rieck, Trinius, Willems, and Holz (2011) adopted a semi-supervised approach to detect obfuscated code. Obfuscated code fragments with similar behavior are grouped using clustering and finally labeled the groups. Next, unknown malware are classified using already labeled groups. Wang, Cai, and Wei (2016) use deep features extracted by Stacked Denoising Autoencoders (SdA) for detection of obfuscated code. Results are promising although SdAs are slow in training. O'kane, Sezer, and McLaughlin (2016) present a method using an optimal set of operational assembly codes (opcodes) from (Ragkhitwetsagul, Krinke, & Clark, 2016) who perform pervasive modifications of source code using the ARTIFICE tool (Schulze & Meyer, 2013) and bytecode obfuscation using the Pro-Guard tool.[1] Pro-Guard optimizes Java bytecode and uses reverse engineering by obfuscating the names of classes, fields and methods. The modification includes changes in layout and renaming of identifiers, changes that affect the code globally. Local modification of code is also performed. They use compilation and decompilation for transformation (obfuscation) and normalization, respectively.

Despite there being a number of tools and methods for detecting code obfuscation, the effectiveness is unclear. The reason is

[1] https://www.guardsquare.com/en/proguard.

that they analyze the structure of the code and pay scant importance to the semantics or meaning of the code. As a result, structural variations of the code remain undetectable using the above methods. Because of the resemblance of the problem to semantic code clone detection, we model it as a code clone detection problem.

Our current study is restricted to Java language code only. This is because, for almost 25 years, Java is a top programming language (Krill, 14.04.2015) in terms of Tiobe's index. It gains further popularity due to recent advances in Android based applications. Semantic similarity remains challenging to detect for any programming languages and often used by the attackers to create obfuscated codes. It has been noticed that some attackers clone the code from legitimate Android apps and pack or assemble the code with modifications after reverse-engineering. Some studies even focusing on detecting mobile app clones by analyzing bytecodes or opcodes (Chen, Alalfi, Dean, & Zou, 2015; Zhou & Jiang, 2012). Our intention is to show that lower level code features derived from bytecodes (or opcodes for other languages) along with source code features are higher effective in detecting clone or obfuscated code. Extraction of bytecodes is another potential reason for selecting Java for our study. We believe that certain aspect of semantics can be felicitously captured by bytecode when used with source codes. However, our current framework can easily be extended to other programming platforms.

We propose a single detection framework for both semantic Java code clones and obfuscated Java code using machine learning.

4. An integrated detection framework

A straightforward approach to determine if two fragments of code are semantically similar without necessarily being syntactically similar may proceed as follows: trim and normalize the two fragments as discussed earlier, obtain the formal semantics of the two using a method alluded to earlier, and compare the formal semantic representations using Eq. (2). However, tools to obtain formal semantics are not readily available. In addition, formal semantic representations are strings themselves, requiring additional string comparisons. It is also unclear that formal semantic representations will add substantially to efficient and effective code clone or obfuscated code detection. Thus, it may be appropriate to investigate if other approaches may work well in detecting if two code fragments are semantically similar with each other, and additionally if they are obfuscated.

Code clone or obfuscated code detection has been treated as a pairwise similarity analysis problem, where two pieces of code are semantic clones or obfuscated if a given piece of code is semantically similar to the given reference code. However, machine learning usually considers individual samples for training and predicts class labels. Instead of comparing the structural and meaning representations (which may be long and/or difficult-to-obtain strings themselves) directly, to compare if two code fragments are syntactically or semantically similar, we can extract relevant characteristics of the code fragments by looking at selected portions of them or other associated structures; such characteristics are usually called *features* in the machine learning literature. To apply machine learning to pairwise clone detection, we use features of both the reference and target code fragments.

Definition 5 (Pairwise Learning)**.** Given a set of N pairs of training samples, each sample (a pair of code fragments) labeled with a clone type or agreement about obfuscated or non-obfuscated codes depending on their mutual similarity, a classification model can act as a mapping function f: $X \rightarrow Y$, where X is an unknown pair of code fragments and Y is the possible clone type (or, obfuscated code) predicted by the model. Training samples are represented as feature vectors, $features(\langle \mathcal{C}_i, \mathcal{C}_j \rangle) = \langle f_1, f_2, \cdots, f_M, D_k \rangle$ of size M, created by combining the features of two different pieces of codes $(\mathcal{C}_i, \mathcal{C}_j)$ and a binary decision variable or class, D_k associated with $(\mathcal{C}_i, \mathcal{C}_j)$, forming a training sample matrix of size $N \times (M+1)$.

The similarity between two code fragments is measured by computing similarity between the two feature based representations. The relevant features for a pair of code fragments can come from many sources. One such source is bytecode Dependency Graph (BDG) representation. Java bytecode representation is less ambiguous than high-level source code. In our work, we use broadly two categories of code fragments features, bytecode or low level features; and source code or high level features. Examples of features we use in our work are given below.

- Low Level Features: Bytecode (BC) features and Bytecode Dependency Graph (BDG) features.
- High Level Features: Traditional features, Abstract Syntax Tree (AST) features, and Program Dependency Graph (PDG) features.

Next, we discuss in detail about various code fragment features we use for semantic clone or obfuscated code detection.

4.1. Java bytecode: low level features

Java source programs are compiled into a portable binary format called bytecode. The bytecode is an intermediate program between Java source code and machine code. The Java bytecode is a sequence of instructions for the virtual machine to execute basic functionalities such as conditions and loops. Each bytecode contains one or more opcodes. The Java Virtual Machine takes the bytecode and converts it into machine code. When the Java Virtual Machine (*JVM*) loads a class file, it is executed by an interpreter. This file contains a stream of bytecodes instructions for each method in the class. Bytecode is a low-level representation for Java programs and hence is likely to be effective for representing the semantics of a program.

Although we work with Java code only, we can generalize our method to convert code in any language into binary code and appropriate intermediate languages whenever possible to detect code clones or obfuscations. For example, Microsoft .NET programming languages can be converted into the Microsoft Intermediate Language (MSIL). We attempt to represent the meaning of a program by extracting interdependency relationships among different bytecode constructs. We represent such dependency as a graph called Bytecode Dependency Graph (BDG). An illustration of the BDG construction scheme is depicted in Fig. 1. BDGs represent both data and control dependencies for each operation in the bytecode. We create a BDG from the bytecode and extract features from the graph. The BDG features are the *semantic* or meaning features. We extract control dependency features by reading the *.class* file sequentially and by tracking down all the instructions that may cause conditional or unconditional branching of the control flow of the code sequence. We consider three types of control instructions, which are listed in the Table 1. We find the frequencies of various data and control dependency relationships among different instructions. We consider a total of 23 constructs and 85 relationships between them and use them as our BDG.

We feel that similarity between two blocks of bytecode can be used as a measure of similarity between the semantics of a pair of original source code fragments. We pre-process the source code by trimming and normalizing as discussed earlier. We first extract syntactic features from the program bytecode. Frequency of occurrence of various bytecode instructions such as *load, store, add*, and *sub*, are used as features, what we call Bytecode features. When parsing the *.class* file, we ignore certain bytecode entities like statement numbers. Such low information entities are unlikely to contribute to the meaning of a program and hence we remove them.

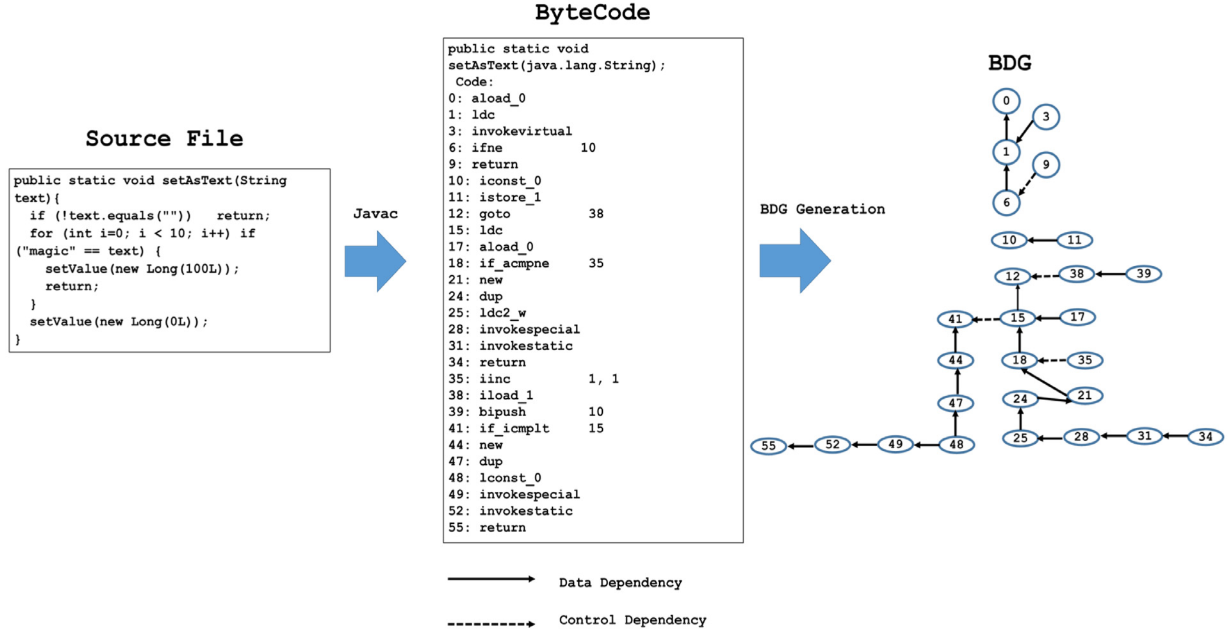


Fig. 1. A BDG showing control and data dependency among the instructions.

Table 1
Bytecode conditional satatements.

Control	Instructions
Unconditional branch	goto, goto_w
Conditional branch	ifeq, iflt, ifle, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpgt, if_icmpge, if_acmpeq, if_icmplt, if_icmple, if_icmpne, if_acmpne
Compound cond. branch	tableswitch, lookupswitch
Comparisons	lcmp, fcmpg, fcmpl, dcmpg, dcmpl

Table 2
Categorization of bytecode instructions.

Category	Instructions
Load	aload, dload, fload, iload, lload
Store	astore, dstore, fstore, istore, lstore
const	aconst, iconst, dconst, fconst, lconst
Arithmetic	iadd, dadd, fadd, ladd, isub, dsub, fsub, lsub, imul, dmul, fmul, lmul, idiv, ddiv, fdiv, ldiv, irem, drem, frem, lrem
Type conversion	i2l, i2f, i2d, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c, i2s

We classify the instructions into several categories for our ease of computing the features and dependency relationships. The categories are listed in Table 2.

We extract both bytecode (BC) and BDG features from the BDG itself. Algorithm 1 shows the steps in extracting such features from a BDG. It takes Java code as input and generates the bytecode as a *.class* file using the Javac compiler. $\mathcal{L}_{BC}$ and $\mathcal{L}_{BDG}$ are the vectors of pre-specified BC and BDG attributes, respectively. The algorithm counts the frequencies of the target attributes for both BC and BDG. In case of BC features, `MatchToken` matches each pre-specified BC attribute and increases the count of the frequency of the target feature in the $\mathcal{L}_{BC}$ vector. To extract BDG features, the method `DependencyFreq` checks for all possible control dependency relationships that persist in the bytecode. Similar to `MatchToken`, it increases the count if the specified relationship, given as BDG attribute is encountered during each iteration. It returns the frequency vector and stores it as $\mathcal{L}_{BDG_i}$. Frequencies of BC and BDG attributes are finally stored into a vector F as features of a given Java program. Please refer to Supplementary materials for the details of the features we extract as BC and BDG features.

Algorithm 1 Bytecode & BDG Feature Extraction

1: **INPUT** : $\mathcal{C}$ // Java Source Code
2: **OUTPUT** : $F = \{f_{BC_1}, \cdots, f_{BC_N}, f_{BDG_1}, \cdots, f_{BDG_M}\}$ // Set of N BC and M BDG features
3: **Steps** :
4: $\mathcal{L}_{BC} = \{B_1 \cdots B_N\}$ // List of N BC attributes
5: $\mathcal{L}_{BDG} = \{D_1 \cdots D_M\}$ // List of M BDG attributes
6: $\mathcal{T} \leftarrow$ `javac`(C) //invoking Javac compiler
7: $\mathcal{V} \leftarrow$ `Tokenize`$(\mathcal{T})$ //Read line by line the .class file and store the stream sequence of instructions in a vector $\mathcal{V}$
// Counting frequency of Bytecode instructions features
8: **for** $i = 1 \cdots |\mathcal{L}_{BC}|$ **do**
9: **for** $j = 1 \cdots |\mathcal{V}|$ **do**
10: **if** `MatchToken`$(B_i, \mathcal{V}_j)$ **then**
11: $f_{BC_i} = f_{BC_i} + 1$
12: **end if**
13: **end for**
14: $F = F \cup f_{BC_i}$
15: **end for**
//Counting frequency of BDG features
16: **for** $i = 1 \cdots |\mathcal{L}_{BDG}|$ **do**
17: $f_{BDG_i} \leftarrow$ `DependencyFreq`$(D_i, \mathcal{V})$
18: $F = F \cup f_{BDG_i}$
19: **end for**
20: **return** F

4.2. Source code features

Besides using low level bytecode for capturing the semantics of a Java code snippet we also extract semantic as well as syntactic features directly from the high level source code. The combination of both low and high label features may represent the semantics

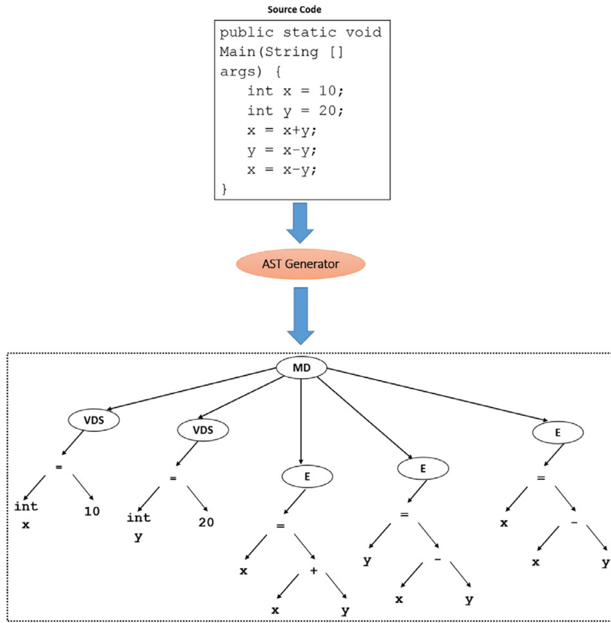


Fig. 2. Example of AST derived from code block. *MD: MethodDeclaration VDS: VariableDeclarationStatement, CE: ConditionalExperssion, E: Expression.*

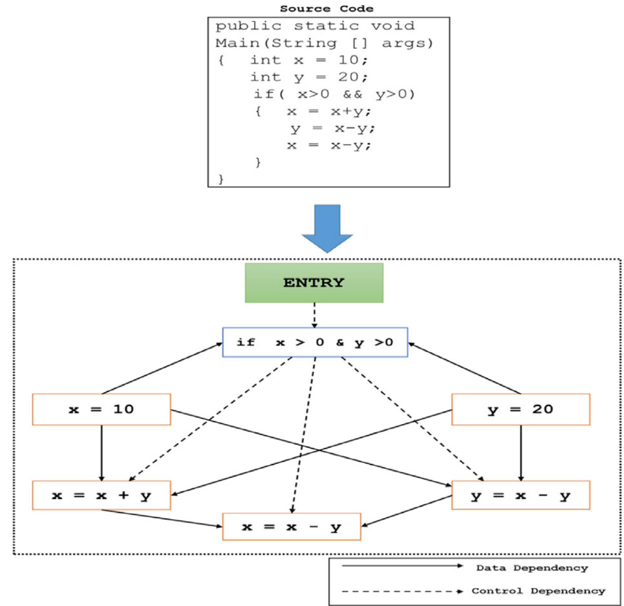


Fig. 3. Program dependency graph showing control and data dependency among the statements.

and syntax more accurately, hence, helping in better matching of two target programs.

Syntactic similarity between two code blocks is also likely to impact upon the similarity in meanings, and hence we also parse the code fragments into their structural components in terms of Abstract Syntax Tree (AST) (Baxter, Yahin, Moura, Sant'Anna, & Bier, 1998). Each node of the tree represents a programming construct occurring in the given source code. Leaf nodes of the tree contain variables. Unlike a majority of published clone detection methods that compare the two syntactic trees directly, we extract certain characteristics or features from the ASTs. Fig. 2 shows an example AST created by the AST Generator software[2] we use. We traverse the AST in post-order manner and extract only non-leaf nodes containing programming constructs such as Variable Declaration Statements (VDS), While Statements, Cast Expressions, Class Instances, and Method Invocations. We represent the frequencies of these programming constructs as AST features in a vector.

We also extract source code control dependency features from Program Control Dependency Graph (PDG) (Ferrante, Ottenstein, & Warren, 1987). The PDG features are a type of *semantic* features. PDGs make explicit both the data and control dependence for each operation in a program. Data dependencies represent the relevant data flow relationships within a program. Control dependencies represent the essential control flow relationships. A sample PDG derived from a code fragment is given in Fig. 3. Edges represent the order of execution of program nodes. Nodes represent the lines where the corresponding elements are located in the program. Horwitz, Prins, and Reps (1988) show that PDGs can be used as "adequate" representations of programs and prove that if the PDGs of two graphs are isomorphic, they are strongly equivalent, i.e., they are "programs with the same behavior." We extract features, instead of directly matching the two PDG graphs for clone detection. Examples of such features are the number of Assignments that come after Declarations, obtained by counting the occurrence of the assignments which are dependent on declarations; the number of Declarations coming after Control (e.g. i < count, for, while, if, switch etc.), the numbers of times nested iterations occur; the numbers of times nested selections occur, and so on.

In addition to all of the features discussed above, we extract basic source code characteristics, which we call Traditional Features. They include the number of Lines of Code (LOC), the number of keywords, variables, assignments, conditional statements and iteration statements (Kodhai, Kanmani, Kamatchi, Radhika, & Saranya, 2010) used in a given piece of source code.

4.3. Fusion of code features

We combine feature vectors (Eq. (4)) extracted from a pair of target and reference code codes to create the training dataset.

$$[\![\mathcal{C}_i^n]\!] \approx \langle f^l_{i1},\cdots f^l_{ik_b} \mid f^l_{i1},\cdots f^l_{ik_d} \mid f^h_{i1},\cdots f^h_{ik_t} \mid f^h_{i1},\cdots f^h_{ik_a} \mid f^h_{i1},\cdots f^h_{ik_p}\rangle \quad (4)$$

In this equation, we denote the different categories of features with different superscripts: l for low level or byte code related features, and h for high level source code features. We denote the different types of features with different superscripts: b : bytecode, d: BDG, t: traditional, a: AST and p: PDG. Features are separated into five different groups with vertical lines, for clear separation.

We fuse the sequence of features from the two different codes. Although there are five type of features covering both low and high level features in the description of a code fragment, to simplify the notation, we can rewrite Eq. (4), without distinguishing among the feature types, as:

$$[\![\mathcal{C}_i^n]\!] \approx features(\mathcal{C}_i) = \langle f_{i1},\cdots f_{ik}\rangle. \quad (5)$$

where $k = k_b + k_d + k_t + k_a + k_p$. Similarly,

$$[\![\mathcal{C}_j^n]\!] \approx features(\mathcal{C}_j) = \langle f_{j1},\cdots f_{jk}\rangle. \quad (6)$$

We use known pairs of cloned or obfuscated code fragments for feature extraction and labeling of the class of the feature vector as true clone or obfuscation type.

Given two code fragments $\mathcal{C}_i$ and $\mathcal{C}_j$, and the corresponding class label D for the code fragments, the combined feature vector, $features(\langle\mathcal{C}_i,\mathcal{C}_j\rangle)$ can now be represented as a fused feature vector. We fuse the two vectors in three different ways as discussed below.

[2] http://www.eclipse.org/jdt/.

Linear combination: We concatenate the two feature vectors. Simple concatenation gives rise to a fused feature vector of size $2k$. Linear combination looks like as follows:

$$features(\langle \mathcal{C}_i, \mathcal{C}_j \rangle) = \langle f_{i1}, \cdots, f_{ik}, f_{j1} \cdots, f_{jk}, D \rangle. \quad (7)$$

A linear combination results in double the number of features. To reduce the size, we may use two other simple combination approaches.

Multiplicative combination: Here we combine two different feature sequences by multiplying the corresponding feature values.

$$features(\langle \mathcal{C}_i, \mathcal{C}_j \rangle) = \langle f_{i1} * f_{j1}, \cdots, f_{ik} * f_{jk}, D \rangle, \quad (8)$$

Distance combination: Nearness, the opposite of distance, is the most obvious way to calculate the similarity between two code features. We use the absolute difference between two feature values to fuse the features of a pair of code fragments.

$$features(\langle \mathcal{C}_i, \mathcal{C}_j \rangle) = \langle |f_{i1} - f_{j1}|, \cdots, |f_{ik} - f_{jk}|, D \rangle. \quad (9)$$

4.4. Code similarity as a feature

We use the text similarity score between a pair of target code fragments as one of the features as well. We compute text similarity between two target Java source code fragments $\mathcal{C}_i$ and $\mathcal{C}_j$, and also the text similarity between their corresponding bytecodes $\mathcal{B}_i$ and $\mathcal{B}_j$, respectively. We tokenize each source code fragment and count frequencies of the tokens. We calculate cosine similarity (Candan & Sapino, 2010) using the frequency vectors as follows:

$$\mathcal{S}_1(\mathcal{C}_i, \mathcal{C}_j) = \cos(\theta) = \frac{\sum_{i,j}^{n} c_i \times c_j}{\sqrt{\sum_i^n c_i^2} \times \sqrt{\sum_j^n c_j^2}} \quad (10)$$

where, c_i and c_j are the components of the frequency vectors for $\mathcal{C}_i$ and $\mathcal{C}_j$ respectively. Similarly, we calculate the similarity of two Java bytecode fragments, $\mathcal{S}_2(\mathcal{B}_i, \mathcal{B}_j)$ using the above equation.

Finally, a feature vector for a given pair of code fragments for training can be represented as follows:

$$features(\langle \mathcal{C}_i, \mathcal{C}_j \rangle) = \langle f_{ij1}, \cdots, f_{ijk}, \mathcal{S}_1(\mathcal{C}_i, \mathcal{C}_j), \mathcal{S}_2(\mathcal{B}_i, \mathcal{B}_j), D \rangle. \quad (11)$$

We use a total of 258 features to represent a pair of code blocks. The distribution of features categories we use to train a classification model is shown in Fig. 4.

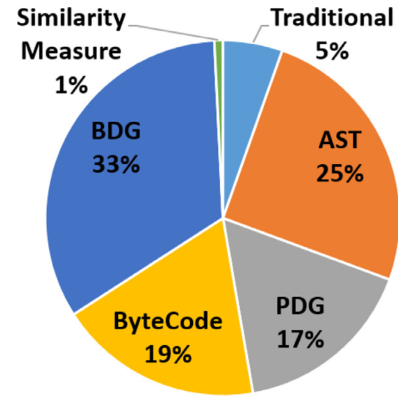


Fig. 4. Share of categories of features used.

4.5. Complexity analysis

We divide our overall tasks of the framework into three parts namely (i) Feature Extraction, (ii) Training and (iii) Testing. The cost of training and testing is model dependent. Since, we are using state-of-the-art models for the purpose, we avoid them while calculating computational complexity. We sub-divide our feature extraction phase into several sub-phases. Phases are listed in Table 3 with their asymptotic cost.

Keeping the dominating factors, out of all the cost and ignoring the rest lower cost terms, the overall complexity for the feature extraction process will be $O(N^3)$ in worst case.

4.6. A new code obfuscation and clone detection scheme

We use our machine learning framework both for detecting code clones as well as obfuscated code. Like any other machine learning framework, our scheme also has two phases, training and testing. In training, we use labeled pairs of cloned code or known obfuscated code from a given corpus. We perform pre-processing steps, including trimming and normalization. Next, we compile the code blocks to Java bytecode classes or files. Then, we generate both low level and high level features from the given pair of code in terms of BC, BDG, AST and PDG features and fuse feature vectors of two target code blocks using one of Eqs. (7), (8) or (9).

Table 3
Computational costs of each sub-functions.

Function	Complexity
Extract Java method blocks	It can be extracted in single scan of the whole file with a cost $O(L)$, where L is the total number of statements in the entire program.
Preprocessing (trimming and normalization)	Similarly, it can be normalized and trimmed in a single scan of the block costing $O(l)$, for l statements per block.
Extract traditional features (per block)	For single scan of the block with n lines costs $O(n)$. So for M blocks it will be $O(M^*n) \approx O(n2)$.
Generate AST	Let h be the height of the AST. Let $N > 0$ be the number of tokens in the tree. Let m be the maximum number of children a node can have. Each node can have at most $m-1$ keys. The complexity comes out to be $O(N)$ (Kelly, 2014)
Extract AST features	Considering worst case scenario of skewed tree, post-order traversal of AST with V tokens or nodes needs , $O(V)$ time. And time for matching each tokens for frequency, it needs $O(N^*V)$, where N is the number of possible AST features. So total comes out to be $O(V) + O(N * V)$. In case where all the tokens are matching with all AST features, it gives to $O(N^2)$ which comes out to be $O(N^2)$ in worst case.
Generate PDG	$O(N)$
Extract PDG features	*DependencyFreq*(P_i, N), searches entire vector of size N for a relation P_i, costing $O(N^2)$. For each P_i in PDG, it will take $O(P^*N^2)$. If number PDG relations are same with numbers of actual relations it will take $O(N^3)$.
Extract bytecode (BC) features	For N number of tokens in ByteCode files it takes $O(N)$ to count the frequency of each BC feature frequency, when we use hashing.
Generate BDG	Similar to AST it costs $O(N)$
Extract BDG features	Extraction of BDG features works similar way like the way we extract PDG features costing $O(N^3)$.
Linear feature fusion	$O(2N) \approx O(N)$, for N features.
Multiplicative feature fusion	$O(N)$
Distance feature fusion	$O(N)$
Code similarity	$O(N)$

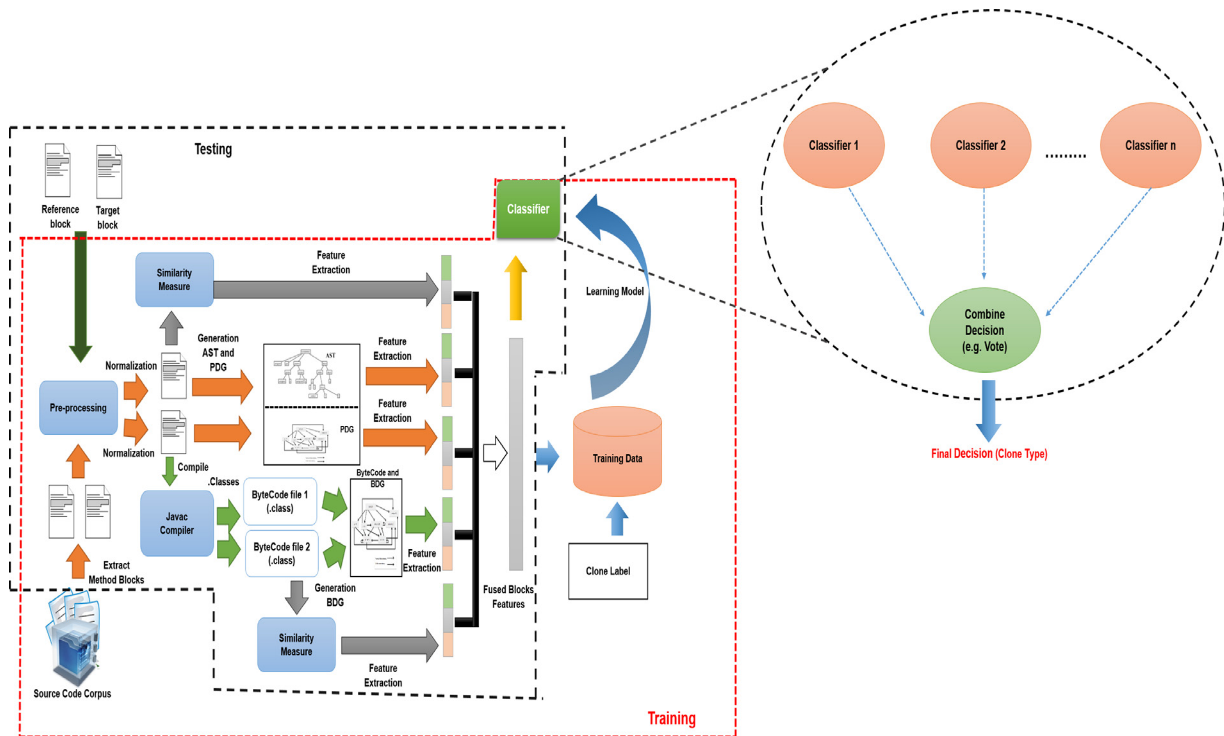


Fig. 5. Workflow of the proposed dual detection framework.

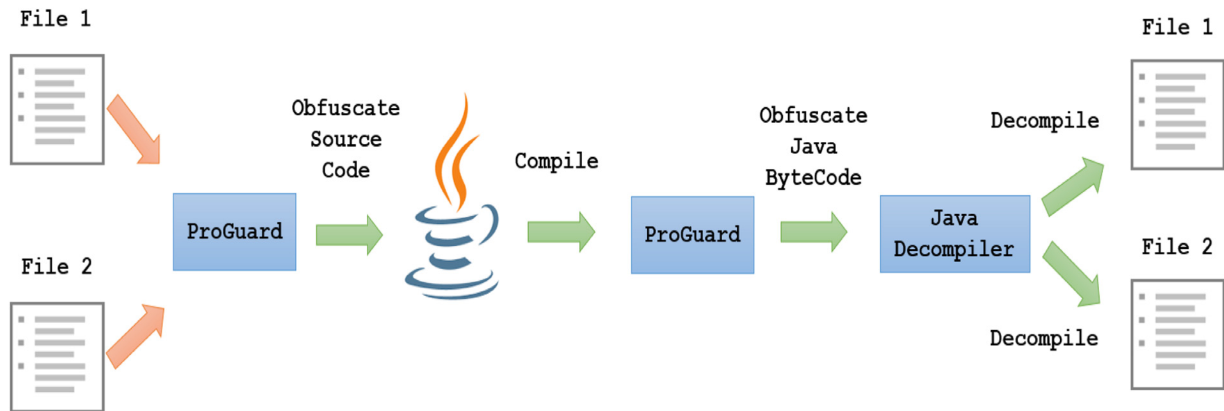


Fig. 6. Java obfuscated dataset generation steps.

We compute similarity between the pair of code blocks using cosine similarity and append them into the fused feature vector. We label the feature vector with a class label based on clone type or whether it is obfuscated code or not. We use a binary classifier for detection of semantic clones or possible obfuscated code. Accordingly, we mark it as Y on N to indicate semantic clone or obfuscated code.

All the above steps are iterated for all possible pairs of code to create a training dataset for the classification model. To identify possible clone or obfuscation in a pair of unlabeled code blocks, we perform the same sequence of steps to create a fused feature vector of the two given blocks and pass it through the classifier for prediction of the possible clone type or to determine if one is an obfuscated version of the other. Fig. 5 demonstrates the work-flow of our approach. We also explore the use of a classifier ensemble using the majority voting approach (Dietterich, 2000) with the hope of achieving better detection rate. It is considered a simple, stable and effective scheme and usually produces more accurate results than a single model. It is known to reduce model bias and variance (Kim, Min, & Han, 2006; Tsai & Hsiao, 2010) to produce more accurate solutions than a single model.

5. Experimental evaluation

We must note that all method blocks of code must be successfully compiled by the compiler suite for our approach to work. If a method block cannot be successfully compiled, it must be corrected manually for further processing. Our approach relies on the Javac compiler. We evaluate the performance of our proposed detection framework in the light of various publicly available obfuscated code and clone datasets. We compare its performance against state-of-the-art obfuscated code detectors and clone detectors. We report them in two different subsections below. In our experiments, we use only examples of Java source code as a corpus for training and testing. However, this model is general in nature and can be extended easily to any other high level programming language. Our primary goal is to improve semantic clone as well as obfuscated code detection accuracy. We use an ensemble of

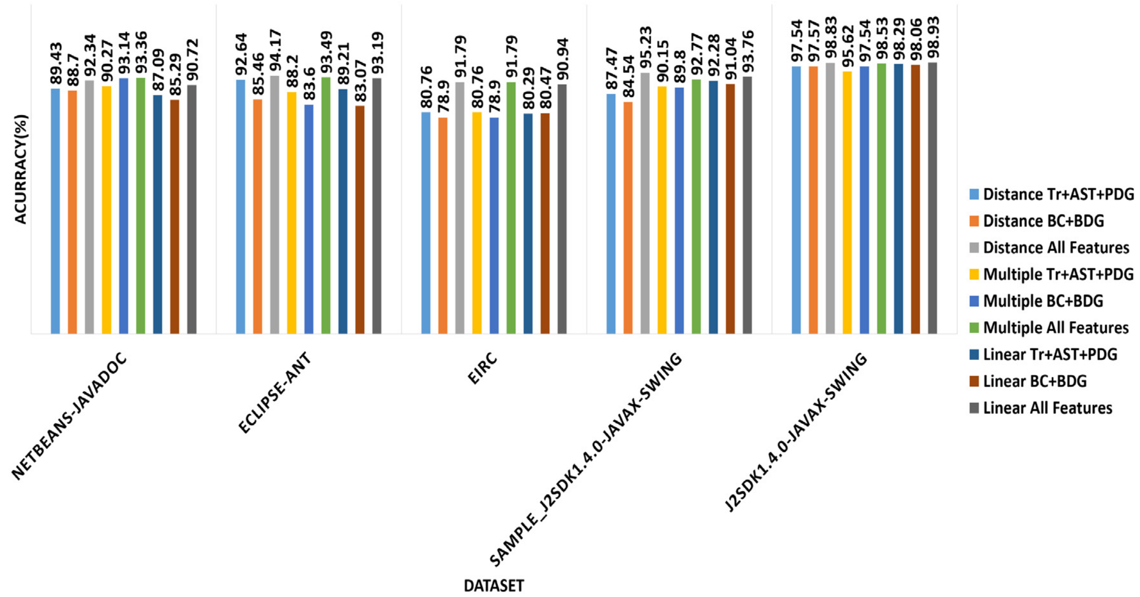


Fig. 7. Performance of detection framework on clone datasets with different features combinations.

Table 4
Brief description of Java obfuscated datasets.

Dataset	Paired codes	Original	Obfuscated
ObsCode	2500	500	2000
PacMan	8464	7968	496
Algorithms	2230	2000	230

selected classifiers and compare the effectiveness of the proposed framework with the state-of-the-art clone and obfuscated code detection methods. We first describe the collection and generation procedure for datasets for different experiments to evaluate the performance of our proposed scheme in detecting both obfuscated code and code clones. We developed a clone labeling scheme using supervised and unsupervised approach with the help of Java programming experts. All our clone datasets are labeled using above schemes. The detail scheme is reported in Sheneamer, Hazazi, Roy, and Kalita (2017).

We evaluate the performance of our proposed detection framework in the light of various publicly available obfuscated code and clone datasets. We compare its performance against state-of-the-art obfuscated code detectors and clone detectors

Next, we report experimental results produced by our scheme with varying combination of feature fusion techniques discussed above. We also train our framework by selecting relevant features from the feature vectors and report the same. Finally, we compare the performance of our scheme with the state-of-the-art detectors.

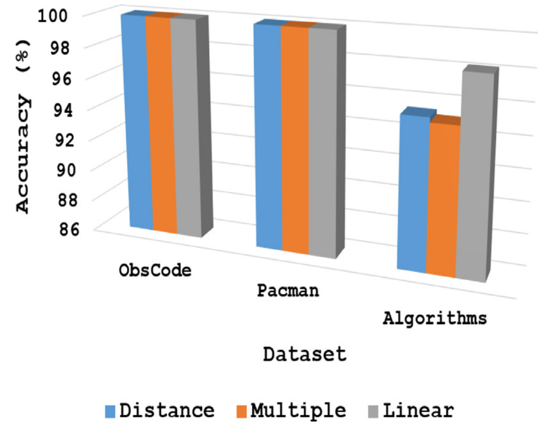


Fig. 8. Effectiveness of the framework on detecting obfuscated code using feature fusion.

5.1. Assessment design

We execute the framework in a similar way for both obfuscated code and code clone detection. First, we train the framework with appropriate samples. To prepare training samples we extract method blocks from known obfuscated code samples or code clones. We extract low level and high level features as discussed earlier and fuse them using the scheme discussed in Section 4.3. We then train a group of classifiers for use as an ensemble to achieve better classification accuracy. We follow the same steps for prediction of unknown samples.

Table 5
Brief description of our Java code clone corpus.

Dataset	Paired codes	Type-I and II	Type-III	Type-I V	False	Agreement (%)
Suple	152	30	59	25	38	75
netbeans-javadoc	452	54	146	39	213	62
eclispe-ant	787	118	392	66	211	60
EIRC	870	32	394	146	298	76
Sample_j2sdk1.4.0-javax-swing	800	200	282	118	200	76
Sample_eclipse-jdtcore	800	200	200	169	231	69

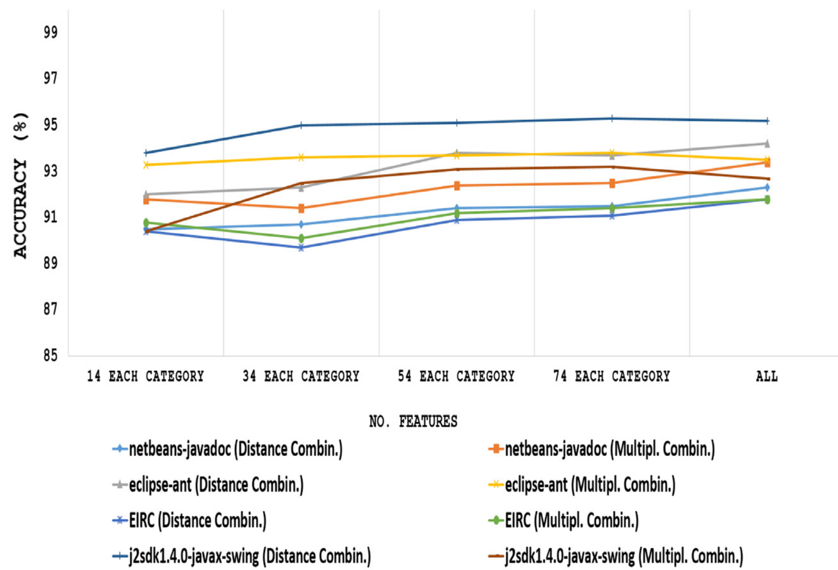


Fig. 9. Learning curve: Performance of ensemble classifier on clone dataset with varying numbers of features.

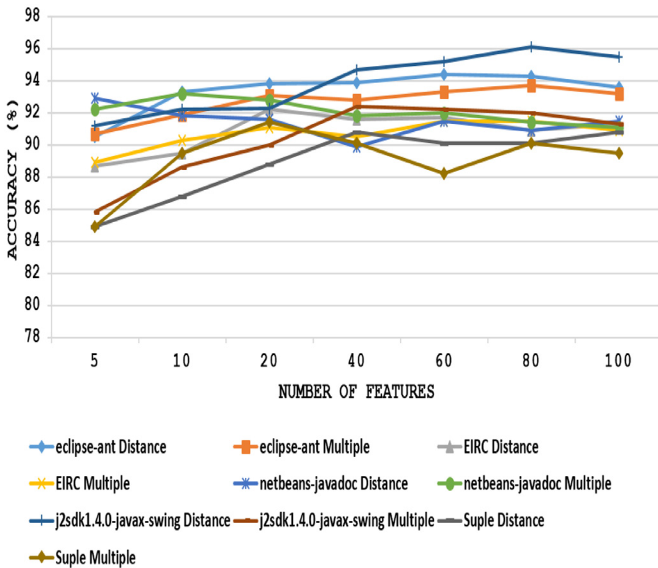


Fig. 10. Performance of ensemble approach on clone dataset after feature selection using *MDI*.

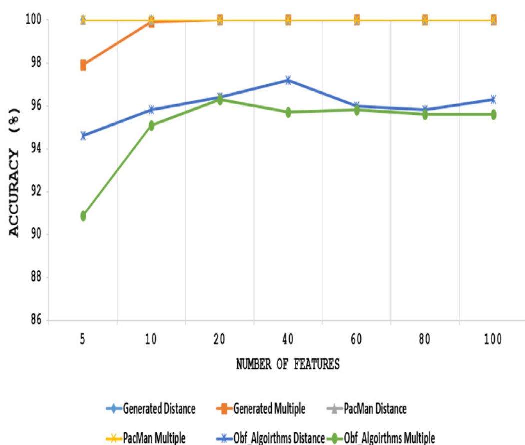


Fig. 11. Effectiveness of ensemble classifier on detecting obfuscated codes after feature selection using *MDI*.

5.1.1. Obfuscated code dataset

We generate examples of obfuscated code using available obfuscation tools. We use five Java classes namely *InfixConverter, SqrtAlgoritm, Hanoi, EightQueens*, and *MagicSquare* to generate Java obfuscated code and name the entire set of classes ObsCode. All of these classes are less than 200 lines of code. Each class of Java code is obfuscated using Artifice (Schulze & Meyer, 2013). Then, the original and obfuscated files are compiled to bytecode. Both byte code files are obfuscated further using ProGuard to create stronger obfuscation. After that, all four bytecode files are decompiled using either Krakatau[3] or Procyon[4] giving back eight additional obfuscated source code files (Ragkhitwetsagul et al., 2016). We generate nine substantially modified versions of each original source code, resulting in a total 50 of files for the dataset (Ragkhitwetsagul et al., 2016).

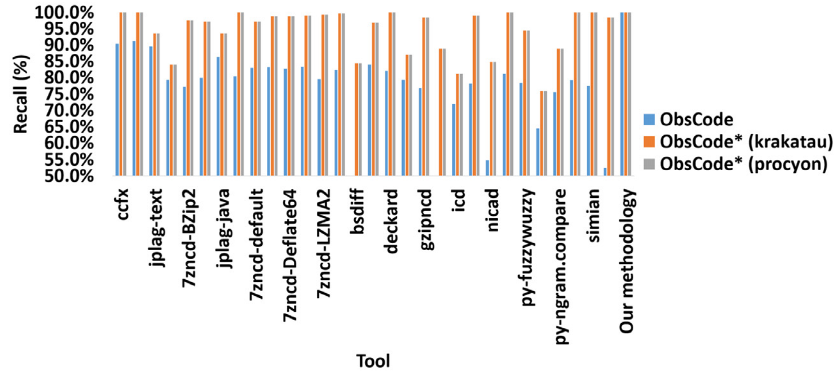
We select the PacMan game[5] as our second subject system (Schulze & Meyer, 2013). It contains of 21 files and 2400 lines of code. The classes are further transformed using renaming, contraction, expansion, loop transformations (Schulze & Meyer, 2013).

We also select supplementary source programs available in the textbook called *Algorithms*[6] containing 149 Java source files and generate the obfuscated code. We generate obfuscation files from the above source files content using the approach illustrated in Fig. 6. Each class of Java code is obfuscated using ProGuard. Then, the original and obfuscated files are compiled to bytecode. Both bytecode files are obfuscated once again using ProGuard. After that, all bytecode files are decompiled using Java decompiler giving back obfuscated source code files. We obtain 785 Java bytecode files for the dataset. A summary of the datasets is given in Table 4.

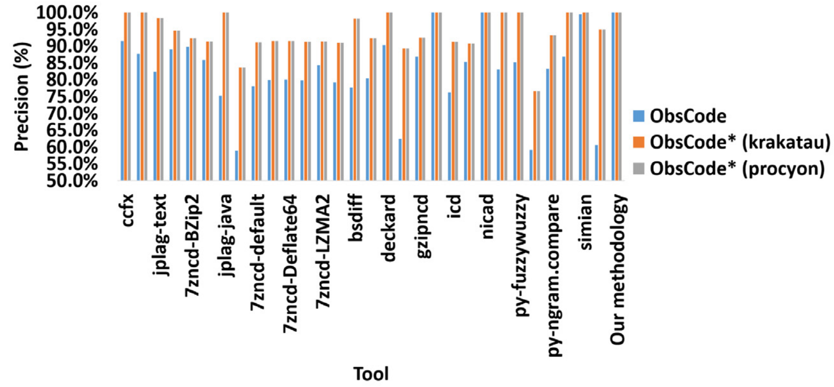
5.1.2. Clone dataset

We use six Java code clone and three obfuscated code datasets for training and testing. Details of the datasets are given in Tables 4 and 5. A majority of existing clone datasets used in prior papers are incomplete in nature. They usually avoid labeling semantic code clones. The publicly available datasets are *eclipse-ant, netbeans-javadoc, j2sdk14.0-javax-swing, eclipse-jdtcore, EIRC* and *Su-*

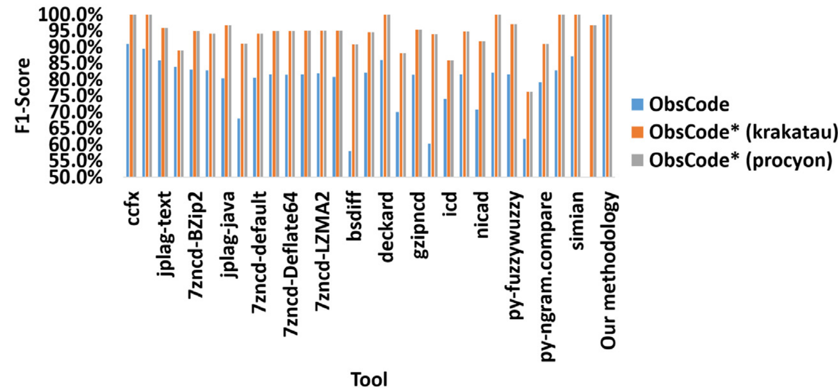
[3] https://bitbucket.org/mstrobel/procyon/wiki/Java.
[4] https://github.com/Storyyeller/Krakatau.
[5] https://code.google.com/p/pacman-rkant/.
[6] http://algs4.cs.princeton.edu/home/.



(a) *Recall*



(b) *Precision*



(c) *F1-Score*

Fig. 12. Effectiveness of various obfuscated code detection tools on ObsCode dataset.

ple. The original datasets contain very small numbers of instances of specific types, making them difficult to use for machine learning. To overcome this situation we extract additional method blocks from the original source programs and label them with the help of a group of expert Java programmers with Masters and PhD degrees in Computer Science. Then, we compute probability of reliability between observers or raters to as certain that labels are acceptable. We compute Kappa statistic (Viera, Garrett et al., 2005) agreement between every two observers' decisions using Eq. (12):

$$\kappa = \frac{p_o - p_e}{1 - p_e}, \qquad (12)$$

where, p_o is the relative observed agreement among raters and p_e is the hypothetical probability of chance agreement.

We finally obtain the average probability of agreement between all the raters for each dataset. A brief summary of the extended datasets is given in Table 5. In the table, the second column indicates how many paired-blocks we extracted to expand the existing dataset. Agreement refers to the probability of reliability between observers or raters. We compute Kappa statistic (Viera et al., 2005) agreement between every two observers' decisions using Eq. (12) and take the average probability of agreement between all the raters and report the same in the table:

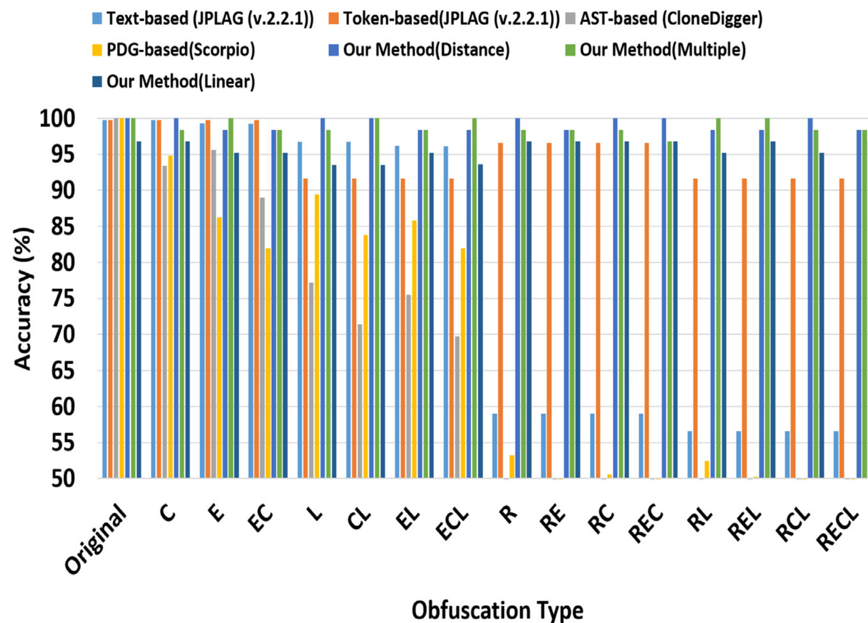


Fig. 13. Tool performance comparison on the PacMan data in terms of accuracy, which original program compared to obfuscated programs for each clone detection technique. The obfuscation are abbreviated as follows: C: contraction, E: expansion, L: loop transformation, R: renaming (Schulze & Meyer, 2013).

5.2. Ensemble classification model

We train and test our model using an ensemble approach using majority voting (Dietterich, 2000) among ten classifiers. We include classification decision from Naïve Bayes (John & Langley, 1995), LibLinear SVM (Fan, Chang, Hsieh, Wang, & Lin, 2008), Instance Based Learner (IBK) (Aha, Kibler, & Albert, 1991), Bagging (Breiman, 1996), Logit Boost (Friedman, Hastie, Tibshirani et al., 2000), Random Committee (Witten & Frank, 2005), Random Subspace (Ho, 1998), Rotation Forest (Rodriguez, Kuncheva, & Alonso, 2006), J48 (Salzberg, 1994), and Random Forest (Breiman, 2001) classifiers and ensemble them based on majority decision to obtain the final class label.

5.3. Experimental results

We generate extensive results to assess the robustness of our proposed model in detecting semantic clones and obfuscated code. We experiment with a varying number of features and with different feature fusions schemes to show that our features are able to achieve a high detection accuracy.

5.3.1. Experimenting with varying feature fusion methods

We assess the importance of combining Traditional, AST, PDG, Bytecode and BDG features and report the results produced by the proposed framework on both clone and obfuscated datasets in Figs. 7 and 8, respectively. Results produced by the ensemble classifier with varying feature fusion methods, show that the performance of the ensemble classifier improves substantially as we combine both syntactic and semantic features to detect clones. Interestingly, the performance of the classifier using semantic features is consistent irrespective of feature types and fusion methods. We also observe that distance and multiplicative combinations produce better results than linear combination for all sizes of data.

In case of obfuscated datasets, the results reported in Fig. 8 show that we achieve 100% accuracy for the first two datasets irrespective of the feature fusion method used. However, for the Algorithm dataset, linear fusion gives better results in comparison to other methods.

5.3.2. Experimenting with selected features

We perform two different kinds of experiments with varying numbers of features, selecting equal numbers of features from each feature type (Traditional, AST and PDG, Bytecode, and BDG) and using a feature selection method (Fig. 10). The intention behind such experiments is to show the significance of our features in achieving better accuracy, and that it is not by chance. The growing learning curve (Fig. 9) clearly indicates that the detection accuracy improves with the increase in the numbers of features.

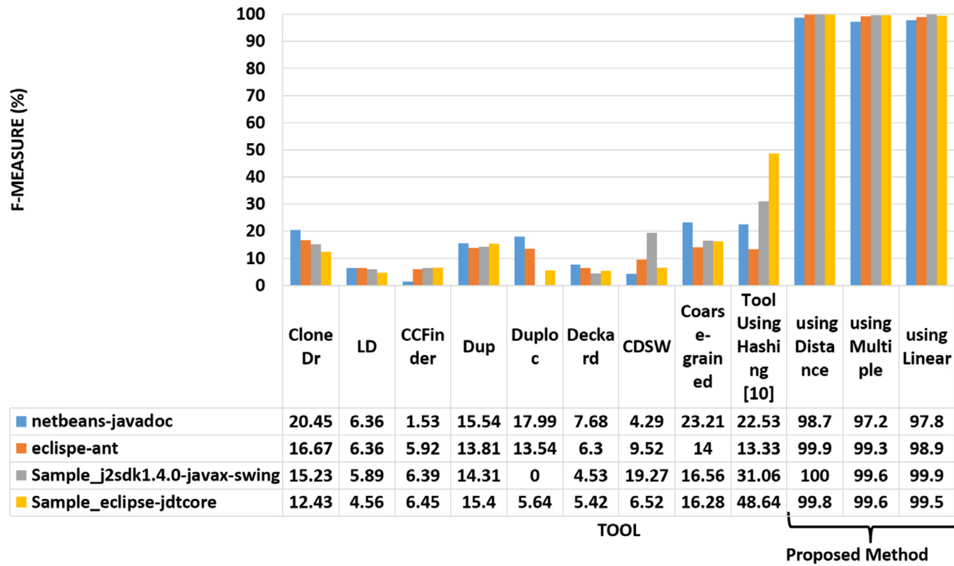
In another experiment, instead of using selection of features from the various categories we use a Random Forests based feature selection algorithm, namely Mean Decrease Impurity (*MDI*) (Louppe, Wehenkel, Sutera, & Geurts, 2013) for selecting feature vectors after applying different fusion methods. Random Forests provide an easy way to assess importance of features based on majority decision using an ensemble of randomized trees. For each experiment, we use different numbers of the feature sets ranked by the feature selection algorithm. Figs. 10 and 11 report the results on clone and obfuscated datasets. Similar to the learning curve based on randomly selected feature sets, growth in the size of selected feature sets shows a growing trend in the performance. This further establishes the fact that our features are crucial in deriving high accuracy detection rates, especially in detecting obfuscated code.

5.4. Performance comparison

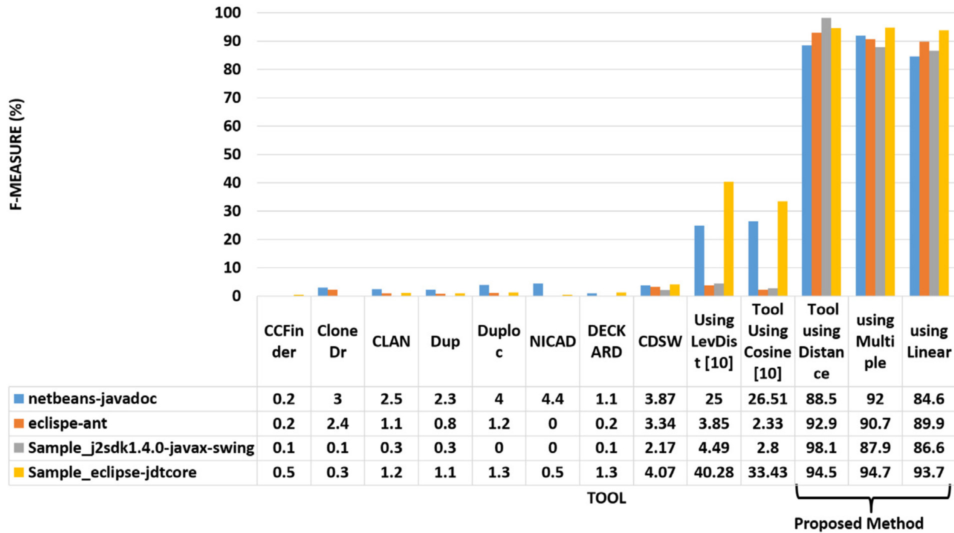
We compare the performance of our method with state-of-the-art clone detectors and contemporary obfuscated code detection tools.

5.4.1. Performance of obfuscated code detectors

We use the previously reported performance scores (Ragkhitwetsagul et al., 2016) in terms of Precision, Recall and F1-score values for comparison of performance of different obfuscated code detectors. We compare our approach with these obfuscated detection approaches: JPLAG, CloneDigger, and Scorpio. We consider the maximum value of the range reported by other authors to give benefit of the doubt to our competitors. Fig. 12 shows a



(a) *Detection rates for Type-I and II clones*



(b) *Detection rates for Type-III clones*

Fig. 14. Prediction effectiveness of proposed framework in comparison to state-of-the-art clone detectors in terms of *F*-score.

comparison of our results with different obfuscated code detectors based on recall, precision and F1-Score on the ObsCode dataset. It is evident that our method performs better than all other methods in detecting obfuscated code. Our method is the winner with the highest F1-Score (100%) in all cases of detecting obfuscated codes based on three different datasets, *ObsCode, ObsCode*(karkatau)*, and *ObsCode* (procyon)*. *ObsCode*(karkatau)*, and *ObsCode* (procyon)* are the variation of our ObsCode dataset created using three different obfuscation tools, Artifice, ProGuard, and Decompilers.

In Fig. 13, we also compare our method with three different obfuscation code detection tools selecting each tool based on the particular detection method they use. In terms of accuracy, our method is the best compared to other four methods, which are Text-based (JPLAG(v.2.2.1)), Token-based (JPLAG(v.2.2.1)), AST-base (CloneDigger), and PDG-based (Scorpio). Our approach achieves 100% accuracy in most of the cases. When we compare our method with other methods for all of obfuscations types, viz, contraction, expansion, loop transformation, renaming, our model is the winner with highest accuracy (98.4%) followed by Token-based(JPLAG(v.2.2.1)) (91.6%), Text-based (JPLAG(v.2.2.1)), PDG-based (Scorpio) (48.8%), and AST-based (CloneDigger) (38.5%), respectively.

5.4.2. Comparison of clone detectors

We compare the performance of our framework with contemporary clone detection methods, using reported results on *eclipse-ant, netbeans-javadoc, j2sdk14.0-javax-swing, eclipse-jdtcore, EIRC* and *Suple* datasets. We compare our approach with several clone detection approaches: CloneDr, CLAN, LD, CCFinder, Dup, Duploc, Deckard, NICAD, CDSW, and Hybrid clone detection. Prior research reports (Hotta et al., 2014; Murakami et al., 2013) report a range of F-scores for detecting Type I, Type II and III only. This is because clone datasets available to them lacked Type IV examples. Moreover, a majority of the detection methods are incapable of detecting semantic clones. Hence, we conduct analysis of the clone detectors for their effectiveness in detecting Type I, II and

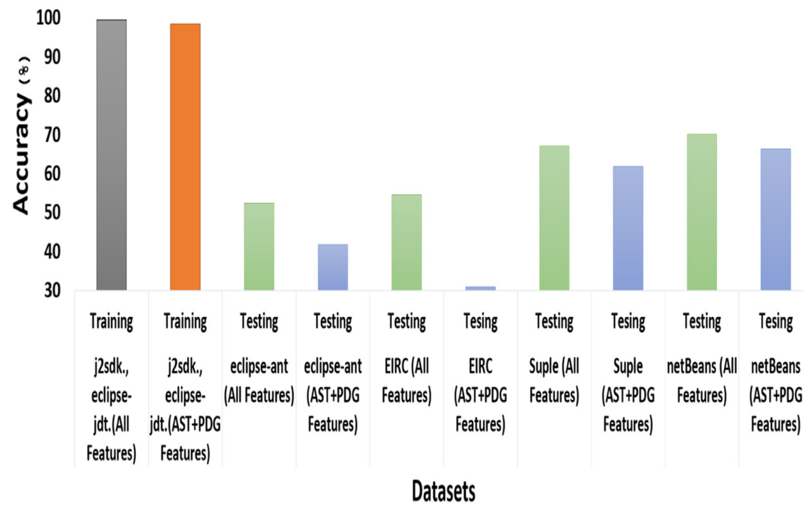


Fig. 15. Effectiveness of the framework for detecting code clones when the training and test datasets are from different corpora.

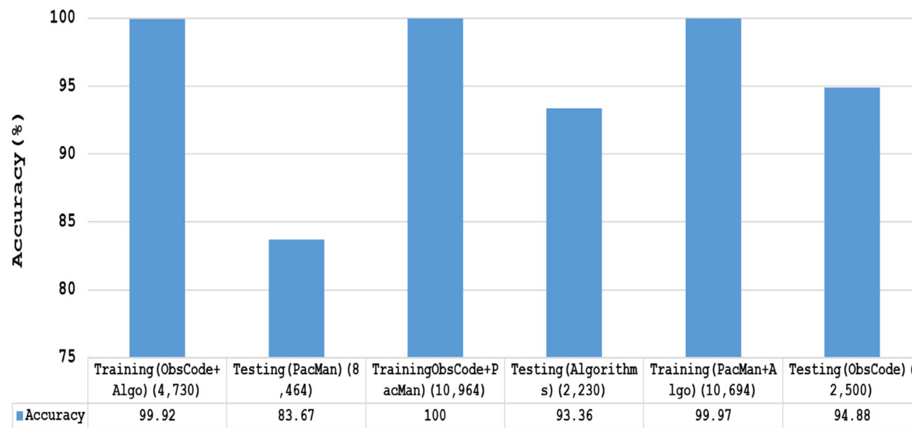


Fig. 16. Performance of the proposed detection framework in detecting obfuscated codes for mix training and testing samples.

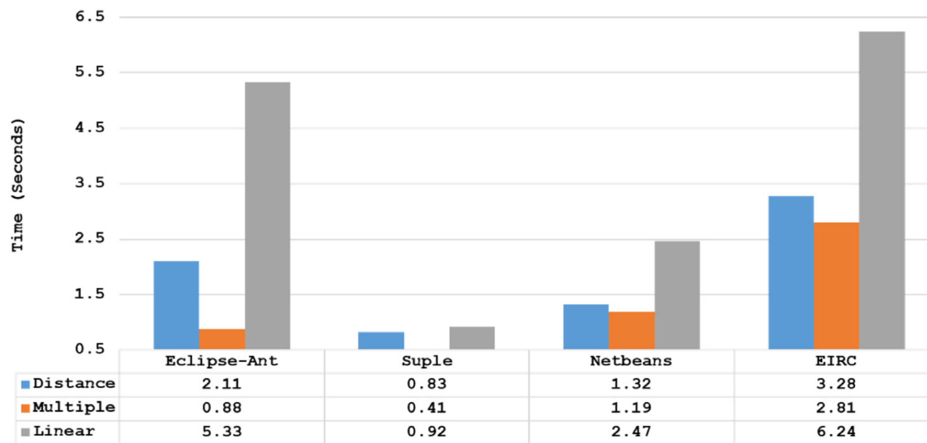


Fig. 17. Execution time requirements on different clone datasets.

III clones. We compare performance of the proposed framework with the contemporary detectors with respect to their maximum reported scores. Fig. 14 shows comparison of our results with the state-of-the-art detectors in terms of F-score. Results clearly establish that our method is superior in detecting all type of clones. Our primary goal is to improve clone detection accuracy for Type-III and Type-IV clones using an ensemble approach using majority voting (Dietterich, 2000) among ten classifiers. We conduct 5 sets of experiments of code clone detection. Each set contains 3 different categories of pair instance features using the three composition functions. Models of the classifiers are produced and tested using cross-validation with 10 folds, where we ensure that the ratio between match and non-match classes is the same in each fold and the same as in the overall of each dataset. Results for only 5 sets of experiments are given in Fig. 7. Fig. 7 shows the accuracy of different code clone detectors based on feature pair instance

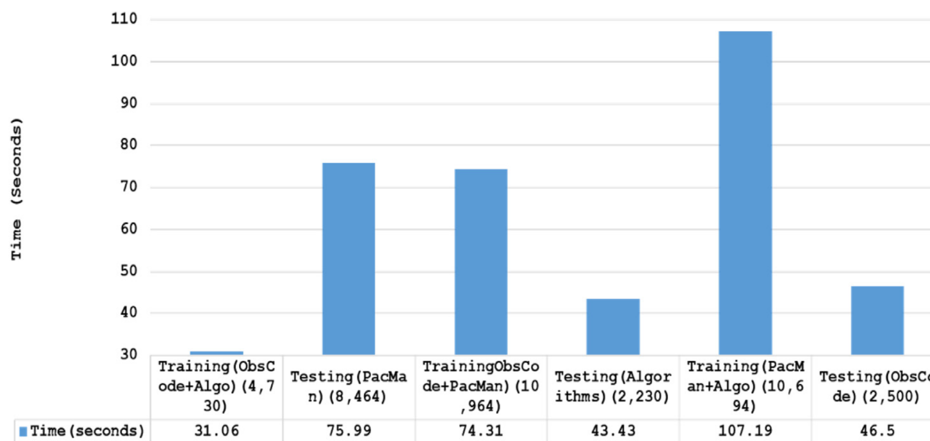


Fig. 18. Time taken by the proposed model for training and testing on different obfuscated dataset combinations.

vectors by the three composition functions. On average, the performance improves by 3.15% when syntactic features and semantic features are added in terms of accuracy in netbeans-javadoc, 6.59% in eclipse-ant, 11.49% in EIRC, 3.84% in Suple, 4.71% in j2sdk1.4.0-javax-swing, and 2.46% in eclipse-jdtcore. This proves our hypothesis that adding more complex features beyond those used traditionally is extremely helpful in code clone detection.

5.5. Extended experiments

Ideally, a detection framework should work on other sample that are different from the samples used during training phase. To evaluate the clone detection capability of our framework in mix training and testing samples, we integrate *j2sdk14.0-javaxswing* and *eclipse-jdtcore* systems and use the integrated dataset as a training set for building the model and estimate the parameters of the model during the learning phase. We then use a different and completely new dataset for model evaluation. The accuracy of the framework in clone detection based on mixed samples is shown in as in Fig. 15.

Similarly, we mix training and testing samples from different obfuscated datasets for detecting code obfuscation using three different ways. 1) Combining *obsCode* and *Algorithms* dataset and evaluate using the *PacMan* dataset, 2) combined *obsCode* and *PacMan* dataset and evaluate using the *Algorithms* dataset and lastly, 3) we combine *PacMan* and *Algorithms* dataset and evaluated using the *ObsCode* dataset. The results from combined datasets are shown in Fig. 16. In terms of accuracy, our obfuscation code detection model produces better results even when the model is built with a dataset and tested on another.

5.6. Execution time performances

We measure the execution time of our approach on different code clones datasets. Fig. 17 shows the execution time obtained by varying steps or task to detect clones in the dataset. Our approach can detect clones in a less than 1 s to a few seconds. Also, we measure our approach on different obfuscated code datasets. Fig. 18 shows the execution time obtained by varying steps or tasks to detect obfuscated code in the dataset. Our approach can detect obfuscated code in about 31 s to about 2 min. Therefore, our approach can detect both clones and obfuscation of code in a short time. Our approach can scale to process millions of files with billion lines of code in a reasonable amount of time. This may prompt new techniques for large datasets clone detection to appear.

6. Threat to validity

The use of the Javac compiler is limited to the compilation technique of Java platform. Our results might not be applicable to other languages that use some intermediate representation of the source code such as C# and C/C++. All the Java files have to be compiled into Java byte prior generating BDG and extracting its features. Therefore, all Java files are required to have no errors before compiling to Java ByteCode and generating BDG.

7. Conclusion

In general, the semantics of a program is difficult to characterize, especially if the semantic representation is to be used for tasks such as detecting software clones and obfuscated programs. A number of methods and software tools are available for detecting code clones or obfuscated code. In this paper, we proposed a novel framework for detecting both Java code clones and Java obfuscated code. We captured the semantics of program codes using low and high level program features derived from Bytecode, AST and PDG. We performed an extensive set of experiments to show that our framework is able to detect both code clone and obfuscated code equally well. The detailed results we present in this paper and in Supplementary Materials clearly establish the machine learning approach we use with the carefully obtained features as the current best method for simultaneously detecting all four types of clones as well as obfuscated code. The current framework is limited to only Java code. Although we believe the ideas can be applied to other languages early. We are exploring ways to make the framework more general in nature, with an eye to its high commercial importance.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.eswa.2017.12.040.

References

Agrawal, A., & Yadav, S. K. (2013). A hybrid-token and textual based approach to find similar code segments. In *2013 fourth international conference on computing, communications and networking technologies (ICCCNT)* (pp. 1–4). IEEE.
Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning, 6*(1), 37–66.
Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. In *International conference on software maintenance, 1998. Proceedings* (pp. 368–377). IEEE.
Breiman, L. (1996). Bagging predictors. *Machine Learning, 24*(2), 123–140.
Breiman, L. (2001). Random forests. *Machine Learning, 45*(1), 5–32.

Candan, K. S., & Sapino, M. L. (2010). *Data management for multimedia retrieval*. Cambridge University Press.
Chen, J., Alalfi, M. H., Dean, T. R., & Zou, Y. (2015). Detecting android malware using clone detection. *Journal of Computer Science and Technology, 30*(5), 942–956.
Christodorescu, M., Jha, S., Seshia, S. A., Song, D., & Bryant, R. E. (2005). Semantics-aware malware detection. In *2005 IEEE symposium on security and privacy* (pp. 32–46). IEEE.
Collberg, C. S., & Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering, 28*(8), 735–746.
Dietterich, T. G. (2000). Ensemble methods in machine learning. In *International workshop on multiple classifier systems* (pp. 1–15). Springer.
Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., & Lin, C.-J. (2008). Liblinear: A library for large linear classification. *Journal of machine Learning research, 9*(Aug), 1871–1874.
Ferrante, J., Ottenstein, K. J., & Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS), 9*(3), 319–349.
Friedman, J., Hastie, T., Tibshirani, R., et al. (2000). Additive logistic regression: A statistical view of boosting (with discussion and a rejoinder by the authors). *The Annals of Statistics, 28*(2), 337–407.
Gunter, C. A. (1992). *Semantics of programming languages: Structures and techniques*. MIT Press.
Higo, Y., Yasushi, U., Nishino, M., & Kusumoto, S. (2011). Incremental code clone detection: A PDG-based approach. In *2011 18th working conference on reverse engineering (WCRE)* (pp. 3–12). IEEE.
Ho, T. K. (1998). The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 20*(8), 832–844.
Horwitz, S., Prins, J., & Reps, T. (1988). On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages* (pp. 146–157). ACM.
Hotta, K., Yang, J., Higo, Y., & Kusumoto, S. (2014). How accurate is coarse-grained clone detection?: Comparison with fine-grained detectors. In *Electronic communications of the eighth international workshop on software clones: 63* (pp. 1–18).
Hummel, B., Juergens, E., Heinemann, L., & Conradt, M. (2010). Index-based code clone detection: incremental, distributed, scalable. In *2010 IEEE international conference on software maintenance (ICSM)* (pp. 1–9). IEEE.
Jiang, L., Misherghi, G., Su, Z., & Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on software engineering* (pp. 96–105). IEEE Computer Society.
John, G. H., & Langley, P. (1995). Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the eleventh conference on uncertainty in artificial intelligence* (pp. 338–345). Morgan Kaufmann Publishers Inc.
Juergens, E., Deissenboeck, F., Hummel, B., & Wagner, S. (2009). Do code clones matter? In *2009. IEEE 31st international conference on software engineering, 2009. ICSE* (pp. 485–495). IEEE.
Kamiya, T., Kusumoto, S., & Inoue, K. (2002). Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering, 28*(7), 654–670.
Kelly, S. L. (2014). AST Indexing: A Near-Constant Time Solution to the Get-Descendants-by-Type Problem. Dikinson College Honors Theses. Paper 147.
Kim, M.-J., Min, S.-H., & Han, I. (2006). An evolutionary approach to the combination of multiple classifiers to predict a stock price index. *Expert Systems with Applications, 31*(2), 241–247.
Kodhai, E., Kanmani, S., Kamatchi, A., Radhika, R., & Saranya, B. V. (2010). Detection of type-1 and type-2 code clones using textual analysis and metrics. In *2010 international conference on recent trends in information, telecommunication and computing (ITC)* (pp. 241–243). IEEE.
Komondoor, R., & Horwitz, S. (2001). Using slicing to identify duplication in source code. In *International static analysis symposium* (pp. 40–56). Springer.
Krill, P. (14.04.2015). Java regains spot as most popular language in developer index. https://www.infoworld.com/article/2909894/application-development/java-back-at-1-in-language-popularity-assessment.html.
Li, Z., Lu, S., Myagmar, S., & Zhou, Y. (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering, 32*(3), 176–192.
Likarish, P., Jung, E., & Jo, I. (2009). Obfuscated malicious Javascript detection using classification techniques.. In *Malware* (pp. 47–54). Citeseer.
Louppe, G., Wehenkel, L., Sutera, A., & Geurts, P. (2013). Understanding variable importances in forests of randomized trees. In *Advances in neural information processing systems* (pp. 431–439).
Murakami, H., Hotta, K., Higo, Y., Igaki, H., & Kusumoto, S. (2012). Folding repeated instructions for improving token-based code clone detection. In *2012 IEEE 12th international working conference on source code analysis and manipulation (SCAM)* (pp. 64–73). IEEE.
Murakami, H., Hotta, K., Higo, Y., Igaki, H., & Kusumoto, S. (2013). Gapped code clone detection with lightweight source code analysis. In *2013 IEEE 21st international conference on program comprehension (ICPC)* (pp. 93–102). IEEE.
O'kane, P., Sezer, S., & McLaughlin, K. (2016). Detecting obfuscated malware using reduced opcode set and optimised runtime trace. *Security Informatics, 5*(1), 1–12.
Ragkhitwetsagul, C., Krinke, J., & Clark, D. (2016). Similarity of source code in the presence of pervasive modifications. In *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)* (pp. 117–126). IEEE.
Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security, 19*(4), 639–668.
Rodriguez, J. J., Kuncheva, L. I., & Alonso, C. J. (2006). Rotation forest: A new classifier ensemble method. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 28*(10), 1619–1630.
Roy, C. K., & Cordy, J. R. (2007). A survey on software clone detection research. *Queenś School of Computing TR, 541*(115), 64–68.
Roy, C. K., & Cordy, J. R. (2008). Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE international conference on program comprehension, 2008. ICPC 2008* (pp. 172–181). IEEE.
Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming, 74*(7), 470–495.
Saini, V., Sajnani, H., Kim, J., & Lopes, C. (2016). SourcererCC and sourcererCC-I: tools to detect clones in batch mode and during software development. In *Proceedings of the 38th international conference on software engineering companion* (pp. 597–600). ACM.
Salzberg, S. L. (1994). C4.5: Programs for machine learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993. *Machine Learning, 16*(3), 235–240.
Schulze, S., & Meyer, D. (2013). On the robustness of clone detection to code obfuscation. In *Proceedings of the 7th international workshop on software clones* (pp. 62–68). IEEE Press.
Sheneamer, A., Hazazi, H., Roy, S., & Kalita, J. (2017). Schemes for labeling semantic code clones using machine learning. *16th IEEE intl conf on machine learning and applications (IEEE ICMLA17)*. IEEE.
Sheneamer, A., & Kalita, J. (2015). Code clone detection using coarse and fine–grained hybrid approaches. In *2015 IEEE seventh international conference on intelligent computing and information systems (ICICIS)* (pp. 472–480). IEEE.
Smith, T. F., & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology, 147*(1), 195–197.
Tsai, C.-F., & Hsiao, Y.-C. (2010). Combining multiple feature selection methods for stock prediction: Union, intersection, and multi-intersection approaches. *Decision Support Systems, 50*(1), 258–269.
Viera, A. J., Garrett, J. M., et al. (2005). Understanding interobserver agreement: The kappa statistic. *Family Medicine, 37*(5), 360–363.
Wang, Y., Cai, W.-D., & Wei, P.-C. (2016). A deep learning approach for detecting malicious Javascript code. *Security and Communication Networks, 11*(9), 1520–1534.
Winskel, G. (1993). *The formal semantics of programming languages: An introduction*. MIT Press.
Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques*. Morgan Kaufmann.
Yuan, Y., & Guo, Y. (2011). CMCD: Count matrix based code clone detection. In *2011 18th Asia Pacific software engineering conference (APSEC)* (pp. 250–257). IEEE.
Yuan, Y., & Guo, Y. (2012). Boreas: an accurate and scalable token-based approach to code clone detection. In *2012 proceedings of the 27th IEEE/ACM international conference on automated software engineering (ASE)* (pp. 286–289). IEEE.
Zhou, Y., & Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy (SP)* (pp. 95–109). IEEE.