Cybersecurity

RESEARCH

Open Access

# Are our clone detectors good enough? An empirical study of code effects by obfuscation

Weihao Huang[1,2], Guozhu Meng[1,2*], Chaoyang Lin[1,2], Qiucun Yan[1,2], Kai Chen[1,2] and Zhuo Ma[3]

**Abstract**

Clone detection has received much attention in many fields such as malicious code detection, vulnerability hunting, and code copyright infringement detection. However, cyber criminals may obfuscate code to impede violation detection. To date, few studies have investigated the robustness of clone detectors, especially in-fashion deep learning-based ones, against obfuscation. Meanwhile, most of these studies only measure the difference between one code snippet and its obfuscation version. However, in reality, the attackers may modify the original code before obfuscating it. Then what we should evaluate is the detection of obfuscated code from cloned code, not the original code. For this, we conduct a comprehensive study evaluating 3 popular deep-learning based clone detectors and 6 commonly used traditional ones. Regarding the data, we collect 6512 clone pairs of five types from the dataset BigCloneBench and obfuscate one program of each pair via 64 strategies of 6 state-of-art commercial obfuscators. We also collect 1424 non-clone pairs to evaluate the false positives. In sum, a benchmark of 524,148 code pairs (either clone or not) are generated, which are passed to clone detectors for evaluation. To automate the evaluation, we develop one uniform evaluation framework, integrating the clone detectors and obfuscators. The results bring us interesting findings on how obfuscation affects the performance of clone detection and what is the difference between traditional and deep learning-based clone detectors. In addition, we conduct manual code reviews to uncover the root cause of the phenomenon and give suggestions to users from different perspectives.

**Keywords** Clone detection, Obfuscation, Evaluation

## Introduction

Source code clone refers to the existence of identical or similar source code between two or more code segments. Many studies (Duala-Ekoko and Robillard 2007; Livieri et al. 2007; Göde and Koschke 2011) have shown that code cloning widely exists in software development to improve work efficiency. However, code cloning is a double-edged sword that can also bring negative effects. Due to insufficient inspection of open source projects, lots of problematic malicious or vulnerable code flow into downstream projects, of which the unfavorable impact is drastically magnified through code cloning (Monden et al. 2002; Kim et al. 2017). On the other hand, the reuse of open source projects may cause copyright disputes (Wu et al. 2015). Code plagiarism has always been an intractable problem in intellectual property protection. To identify these code clones, there are emerging many studies (Sheneamer and Kalita 2016; Roy and Cordy 2007) to compute the similarity between two pieces of code. If the similarity exceeds a certain degree, they are

*Correspondence:
Guozhu Meng
mengguozhu@iie.ac.cn
[1] SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] Xidian University, Xi'an, China

clone pairs, otherwise not. The detection performance of a clone detector largely relies on the features harvested from code and their representation. Generally, features are extracted and represented in mainly four levels: literal text, token, syntax, and semantics (Ben-Nun et al. 2018; Kuhn et al. 2007). Based on the features, a clone detector employs varying approaches to compute their similarity. Code obfuscation is a technique that transforms a computer program into code representation that are functionally equivalent but difficult to read and understand (Viticchié et al. 2016). It is an effective manner for program developers to protect their code from being stolen. On the opposite side, attackers are also apt to obfuscate their code, either for making malicious code evade from detection (OKane et al. 2011; You and Yim 2010), or for getting out of a charge of intellectual property infringement. Hence, it is necessary to evaluate the effect of obfuscation against clone detectors.

After surveying for papers in software engineering and security published over the past two decades, we find that although some studies (Schulze and Meyer 2013; Ragkhitwetsagul et al. 2016; Meyer and Schulze 2012) evaluate the resistant of the clone detectors to obfuscation, almost all of them only evaluate the similarity between one code snippet before and after obfuscation ($a$ and $a'$ as shown in Fig. 1). However, in real scenarios, attackers may not obfuscate the problematic code $a$ directly, but firstly customize it manually as $b$ according to their own needs and then obfuscate $b$ to $c$. Hence, what should be evaluated is whether clone detectors can find clones between $\langle a, c\rangle$, rather than $\langle a, a'\rangle$. On the other hand, recent years witness the superior ability of deep learning in abstracting the semantics of code, and hence a line of research (Tufano et al. 2018; Nguyen et al. 2009) learns the embedding of code semantics for clone detection. However, none of these studies have assessed the deep learning-based detectors. In addition, the number of clone detectors evaluated is smaller and the evaluation subjects are mostly proposed before 2010. Meanwhile, the way of obfuscation is relatively simple, implementing several simpler code conversion methods, or employing a smaller number of obfuscators, thus the obfuscation strategies are insufficient in both type and number. All the above impedes in-depth findings and the drawn conclusions are correspondingly plain. Therefore, it is desired to conduct a comprehensive and meticulous evaluation, assessing the attack effects of code obfuscation, and identifying the potential risks of clone detectors under real circumstances.

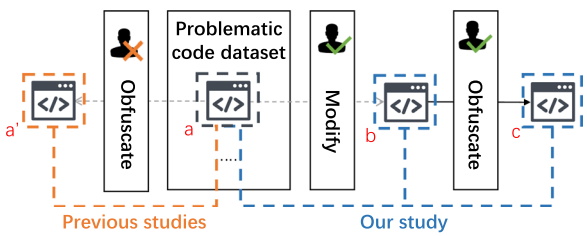


**Fig. 1** The comparison between the evaluation manner of our study and previous studies

In this work, we carefully study the status quo of clone detectors, select representative, state-of-art open-source detectors, including 3 deep learning-based and 6 traditional ones. At the same time, we research the obfuscators in detail and select 6 commercial ones widely used in practice, which basically cover all the four types of basic obfuscation strategies, layout, data, structure, and control flow (Cimitile et al. 2017; Balakrishnan and Schulze 2005; Cimato et al. 2005). Here, we define the first two types as a *simple strategy* and the rest as a *complex strategy*. Furthermore, we devise 69 combinations of compound strategies to measure the superimposed effects between strategies. To simulate the real scenarios, we collect 7,936 code pairs from BigCloneBench (Svajlenko et al. 2014), which contains five types of true clone pairs and false ones, both on function granularity, and employ these obfuscation strategies on them, obtain 428,695 clone pairs and 95,453 non-clone pairs after obfuscation. These samples are then passed to clone detectors for evaluation as our benchmark.

By comparing the change of clone detection performance, we study the effect of obfuscation on clone detectors, including not only obfuscation strategies but also obfuscation tools. We provide cause analysis to explain the essence of these effects. Additionally, we compare the performance of the traditional clone detectors and deep learning-based ones in the consideration of the impact of obfuscation. For ease of evaluation, we construct one unified and scalable framework, which integrates clone detectors and obfuscators freely, obfuscates the code pairs collected from BigCloneBench, passes it to clone detectors for evaluation, and processes the experimental results of various detectors uniformly.

Last, we present a number of insightful findings, including but not limited to: (1) the effect of simple strategies may be greater than the complex ones under some circumstances; (2) the effect of the combined strategies are not necessarily more significant than the basic ones; (3) the deep learning-based detectors are more prone to misclassifying a non-clone pair; (4) for traditional and DL-based detectors, the obfuscation strategy or obfuscation tool having a greater impact on them could be different. (5) The performance of

traditional and DL-based detectors under obfuscation for various clone types is just the opposite. To facilitate future research on clone detection, we will publish our benchmark and evaluation framework at Anonymous (2021) after the work has been accepted.

**Biographytions**. We summarize the contributions as follows.

- **A comprehensive and large-scale evaluation of the attack and confrontation effects between obfuscators and clone detectors.** We construct 69 strategies based on six obfuscators, generating 282,845 obfuscation methods and 524,148 code pairs to evaluate 9 clone detectors, including 6 traditional and 3 deep learning-based tools. *It is noted that we are the first to make an assessment of DL-based detectors under obfuscation.*
- **We construct one benchmark fitting the real scenarios.** As mentioned above, to restore the real scenes, we collect 6512 clone pairs of five types and 1424 non-clone pairs from BigCloneBench and obtain 428,695 clone pairs and 95,453 non-clone pairs after obfuscation through a series of automatic manipulation, which is detailed in Sect. 3, as the benchmark of our evaluation. Furthermore, it could be used for research in related fields and expanded utilizing more obfuscation strategies according to real needs.
- **One open-source and uniform evaluation framework.** We construct an evaluation framework that automates code obfuscation and clone detection. It is extensible and can incorporate more obfuscators and clone detectors easily, expanding the benchmark as needed, making a unified process on the experimental results of various clone detectors according to the performance metrics.
- **Insightful findings and suggestions.** Based on our evaluation results, we conclude a series of findings and suggestions which could be helpful for users from different perspectives, such as designers or analyzers of clone detectors, sparking new thinking in the related research field.

## Background and related work

### Code clone detection

A code clone is a piece of code that is identical or similar to another due to copy and paste programming. To clarify the definition and categories of code clone, (Bellon et al. 2007) first classify them into four categories as per the similarity between clone code pairs. Svajlenko et al. (2014) further refine *Type3* by considering varying degrees of syntactic similarity between code clones.

- $T$ **1** Code is exactly the same except for spaces and comments.
- $T$ **2** Code is syntactically the same, and only variable names, variable types, function types, spaces, and comments are different.
- $ST$ **3** Code is syntactically the same, but there are additions, reductions, or modifications in the statements. The syntactical similarity of code clones locates in the range [0.7, 1.0).
- $MT$ **3** Code is syntactically the same and its syntactical similarity of code clones locates in the range [0.5, 0.7).
- $T$ **4** The functions in code are the same, but the syntax may be significantly different.

In this study, we focus on the clones between Java methods. Therefore, given a set of Java methods $\mathcal{M}$ and clone types $\mathcal{T}$ ($|\mathcal{T}| = 5$), we have a definition for a clone pair as: $\langle m_1,\ m_2,\ t\rangle$ where $m_1, m_2 \in \mathcal{M}$ and $t \in \mathcal{T}$. A false clone pair can be denoted as $\langle m_1,\ m_2,\ \perp\rangle$.

Clone detection is a technique to identify the cloned code through automated similarity comparison and locate their accurate positions (Ain et al. 2019). Given this, it has achieved significant results in malicious code detection, vulnerability hunting, copyright management (Chen et al. 2015; Kim et al. 2017; Liu et al. 2017), and so on. There are a lot of tools and techniques to detect code clones (Jiang et al. 2007; Zhao and Huang 2018; Zhang et al. 2019; Lee and Jeong 2005). In this study, we focus on two types of clone detection as below.

**Traditional detection**

It mainly contains three steps: removing meaningless code, extracting representative features, and calculating the similarity between two code fragments. As for code representation, SDD (Lee and Jeong 2005) constructs text representation from code, CCFinder (Kamiya et al. 2002) and CCAligner (Wang et al. 2018) rely on the token representation. Deckard (Jiang et al. 2007) constructs an abstract syntax tree for code, and Duplix (Krinke 2001) is based on a program dependence graph. Compared with detectors extracting features with text or token representations, the ones based on grammar (such as abstract syntax tree) and semantics (e.g., program dependency graph, control flow graph) can mine deeper information, thereby having the ability to detect more complex clone types such as ST3, MT3, and T4.

**Deep learning-based detection** Recently, researchers apply deep learning to clone detection (Zhao and Huang 2018; Zhang et al. 2019). Compared with the traditional

detector, DL-based detector can further learn the code features thoroughly with layers of networks, thereby digging deeper code semantic information and improving clone detection. In particular, some DL-based clone detectors combine syntax- and semantic-level code representation with DL models, and have obtained superior experimental results. For example, CDLH (Wei and Li 2017) converts the source code into an AST, extracts feature vectors, and then feeds the feature vectors into a customized Convolutional Neural Network (CNN) for training, and finally uses the learned features for similarity analysis. White et al. (2016) use the data obtained by manual sampling as a training set, which is combined with their own defined tree structure and CNN to construct a clone detection model. Zhang et al. (2019) develop ASTNN to solve the problem of gradient disappearance caused by the excessive number of syntax tree layers in the training process by dividing the original abstract syntax tree into smaller subtrees, and combines LSTM for clone detection. DeepSim (Zhao and Huang 2018) generates original features from the control flow graph and data flow graph of the code according to the designed coding method and uses a customized forward neural network model to perform deeper digging of code information at the semantic level.

#### Code obfuscation

Code obfuscation is widely used for intellectual property protection and malicious code hiding. Here, we aim to explore how it influences the detection of code clones. Therefore, we employ four types of basic obfuscation strategies (Collberg et al. 1997) to transform our source code, which are detailed as follows.

- *Layout obfuscation.* In this strategy, we only take into account *identifier replacement* (**IR**). It replaces the name of variables, methods, and classes in the original code with randomly generated strings to eliminate the meaning of specific variables at the source code level.
- *Data obfuscation.* It contains two forms. *Numeric constant replacement* (**NCO**) replaces all numeric constants in the original code with arithmetic operations between multiple numbers, thereby changing the expression of digital constants at the source code level. *String constants encryption* (**SCE**) encrypts all string constants and provides the corresponding decryption function at the same time. When the string constants are needed, the decryption function is invoked to decrypt the strings, changing the form of string constants without changing the program semantics.
- *Structure obfuscation.* There are four forms of structure obfuscation. In particular, *expression replacement* (**ER**) performs the equivalent replacement of expressions in the source code. For example, adding more useless operations makes arithmetic expressions more complicated, thereby increasing the complexity of the original code; *class structure reorganization* (**ISC**) changes the internal structure of a Java class,such as the order of the fields inside one class; *internal class removal* (**ICR**) removes the internal classes defined in one Java class; *code rolling and unrolling* (**CC**) merge several functions into one or split one function into many while retaining the original control flows.
- *Control flow obfuscation.* It adds a number of false conditional control statements to blur the execution logic of code and hinder the comprehension of the original source code (**CFO**). For instance, junk code without any relevance to the original code can be inserted into the code (Cao et al. 2006).

#### Obfuscation on clone detection

Few studies have evaluated the robustness of clone detectors under obfuscation. Schulze and Meyer (2013) studied the effect of 16 simpler obfuscation strategies developed by themselves on three clone detectors. Meyer and Schulze (2012) researched the effect of one obfuscator integrating six obfuscation strategies on three clone detection tools. Ragkhitwetsagul et al. (2016) assessed the performance of five detectors under two obfuscators employing six strategies. All these works mainly focus on presenting the different performance of clone detectors under obfuscation from the perspective of code representation form, such as text, token, etc.. In addition, Ragkhitwetsagul et al. (2016) explores the optimal parameter settings of the clone detection tools. Roy and Cordy (2009) proposes one mutation-based approach for generating clone data, adopting 14 personally defined mutation methods and assessing the performance of only one clone detector.

## Approach

Figure 2 shows the overview of our evaluation work. First, we collect a number of Java code pairs from the benchmark BigCloneBench (Svajlenko et al. 2014). Second, we perform a transformation on every source code to make it compilable and compile it into bytecode. We then build an obfuscation framework that integrates six obfuscators and obfuscates one bytecode of each bytecode pair. The bytecode pair is further decompiled and passed to the clone detection framework, including

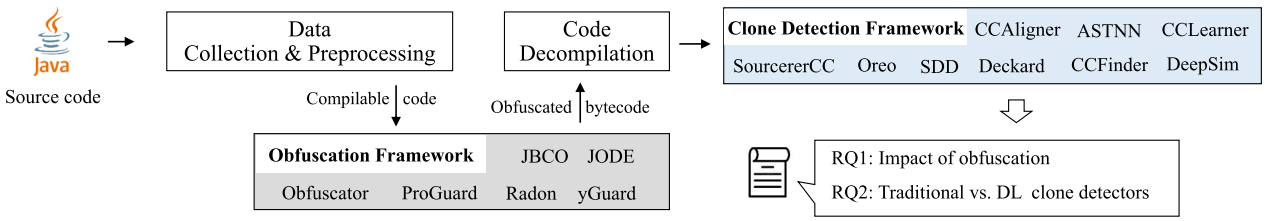


**Fig. 2** Overview of the evaluation work

nine state-of-the-art clone detectors. It is noted that both the obfuscators and clone detectors could be expanded freely. Based on the results of clone detection, we propose two research questions and identify eight findings that characterize the combat between obfuscation and clone detection.

### Data preprocessing

The obfuscation tools only accept bytecode as input, so we need to first make Java methods in the dataset compilable. To this end, we perform static analysis on the target Java methods to: (1) infer the types of local variables (including class fields), the parameters of methods, and the return type of each method; (2) create a dummy class that declares all the missing types of identified objects.

As Algorithm 1, we first use JavaParser (2020) to obtain the abstract syntax tree of the target method. Then we traverse the expressions in this AST (line 2). If the expression contains a type declaration (line 3), we can determine the type of current variable. Otherwise, we check whether there is an implicit relationship between variables (line 4). For example, if one variable $v_2$ is assigned to $v_1$, the type of $v_1$ should be consistent with $v_2$. If there exist any variables whose types are undetermined (lines 8-11), we will query the traversed type relationships. Particularly, if $v$ is of the same type of the undetermined variable *var* (line 9) and $v$ is determined, the type of *var* can be determined thereout. Last, we generate Java classes based on *ctx* (line 12).

**Algorithm 1:** Dummy Class Generation

```
Function Main(ast):
    Input: ast: AST generated by JavaParser
    Output: classes: generated classes
1   ctx ← ∅ // the speculation of variable types;
2   foreach expr ∈ ast do
3       if expr is a type declaration then
4           ⟨var, type⟩ ← inferVarType(expr);
5           ctx ← ctx ∪ ⟨var, type⟩;
6       else
7           {var_i} have same type inferred from expr;
8       end
9   end
10  while there exists var with undertermined type do
11      for v where Type(v) == Type(var) do
12          if v has a determined type then
13              ctx = ctx ∪ ⟨var, Type(v)⟩;
14          end
15      end
16  end
17  classes ← generateClass(ctx);
return classes
```

**Table 1** Obfuscation tools and the corresponding strategies

| Obfuscation tools | Simple strategy | | | Complex strategy | | | | | Stars |
|---|---|---|---|---|---|---|---|---|---|
| | Layout | Data | | Structure | | | | Control-Flow | |
| | IR | NCO | SCE | ER | ISC | ICR | CC | CFO | |
| Radon | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ | 200 |
| JBCO | ✓ | ✓ | | ✓ | | | ✓ | ✓ | 1.5K |
| Obfuscator | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ | 249 |
| JODE | ✓ | | | | | | | | – |
| yGuard | ✓ | | | | | | | | 102 |
| ProGuard | ✓ | | | | | | | | 352 |

### Obfuscation framework

In the framework, we have developed one uniform controller *Wrapper* that can automatically make the required configurations for specific obfuscation strategies and invoke the corresponding tools to obfuscate code in batches, which is convenient to integrate obfuscation tools. Currently, we select six Java obfuscation tools with a high Github star rank. These tools and their supported strategies are summarized in Table 1 and described below.

- **Radon** ItzSomebody (2020) is an open-source small Java bytecode obfuscation tool. This tool obfuscates Java jar packages and supports multiple obfuscation strategies at the same time.
- **JBCO** JBCO (2020) is an obfuscator based on the Soot framework (Soot (2020)). It supports multiple obfuscation strategies to obfuscate jar packages at the same time and achieves stronger obfuscation.
- **Obfuscator** Obfuscator (2020) is an open-source Java bytecode obfuscation tool that supports multiple obfuscation strategies to obfuscate jar packages at the same time and provides a GUI interface to configure obfuscation options.
- **JODE** (Hoenicke 2020) is a decompilation tool and obfuscation tool for Java packages. It supports changing the class name, method name, and field name of Java code to randomly generated ones. Users can provide a conversion table to replace the names of these identifiers.
- **yGuard** yGuard (2020) is a free obfuscator and compressor for Java bytecode. It needs to rely on the Apache Ant tool (2020) ant to run and can be integrated into most commonly used IDEs. It supports the renaming of identifiers in Java bytecode.
- **ProGuard** ProGuard (2020) is currently a well-known optimizer, compressor, and obfuscator for Java bytecode. This tool supports the replacement of identifiers in Java bytecode and the obfuscation of Android applications. It provides a GUI interface to configure various options.

**Obfuscation strategies.** We select all the basic strategies of the six obfuscation tools, and combine different types of strategies of each obfuscation tool, which is a common practice in reality, e.g. (Hammad et al. 2018), obfuscates code with combined strategies to evaluate anti-malware tools. Therefore, we have formed compound strategies that contain either 1, 2, 3, or 4 single strategies. In this manner, we finally obtain 20 basic single obfuscation strategies and 49 combined strategies.

### Code decompilation and clone preparation

We use Procyon (Steiger 2020) to decompile .jar files to obtain Java source code, where the target methods are located. Since the name of the target function may be changed after obfuscation, we recognize it from the main function where we pre-implant a referent to the target method. By traversing all the functions, we manage to extract the target function based on this reference.

To form the benchmark for clone detection, we replace the original methods with their decompiled versions. The reason that we use the decompiled methods is: after decompilation, minor changes may occur in the layout of the original method, such as the replacement of the variables. Therefore, the method we get after obfuscation is not only influenced by the corresponding obfuscation strategies, but also the decompilation process. In order to eliminate this influence from decompilation to make our experiment more accurate, we use the decompiled version of the original method uniformly in both the generation of the obfuscation data set and the original one. Besides, it has been verified that the use of decompiled version could improve the performance of clone detectors (Ragkhitwetsagul et al. 2016; Ragkhitwetsagul and Krinke 2017).

To ensure that the type of clone pairs remain unchanged after decompilation during the generation of the original data set process, we randomly select 100 pieces of records from BigCloneBench covering both the true and false clone pairs and analyze them manually. The analysis result shows that 92 pairs of them retain the original clone type except for individual failures due to the compilation optimization, the percentage of which reaches 92%.

All clone detectors evaluated in this study are binary classifiers, i.e., only distinguish whether the input is a clone or not while not clone types. However, clones of different types may greatly influence detection performance. For a more in-depth study, we create five experimental datasets. More specifically,

we rely on BigCloneBench's labels for clone type and divide the clone pairs into five categories T1, T2, ST3, MT3, T4. Then they are combined with the false clone pairs respectively to form five test subsets. The indicators of detection rate of the five types of clone pairs are calculated to evaluate the effect of the obfuscation on the detection rate of these five clone types.

#### Clone detection framework

In this study, we establish a framework to host clone detectors for evaluation. It provides a consistent dataset for experiments and conducts unified processing of experimental results, comparing the detection rate of the five types of clone pairs respectively before and after obfuscation, and the false clone pairs as well. All these operations are done automatically. For deployment of the clone detectors, what matters most is whether the selected clone detectors are representative and can cover the current mainstream cloning detection techniques commonly used. Although the number of clone detectors is not so large as other types, such as malware detection tools, after decades of research and development, the category and number of it is also considerable, which could be reflected in clone (2020). However these tools have many features in common with each other, thus there is no need to evaluate all of them one by one, resulting in redundancy. To this end, we conduct a comprehensive and in-depth investigation of the source code clone detection work published at academic venues since 2000 and selected nine relatively mature works that are open-sourced as our evaluation objects as shown in Table 2, among which several tools are selected as they are the more classic and influenced ones in clone detection research history, such as SDD, CCFinder, Deckard and the others are relatively new and representative excellent works mainly published on the top conferences or journals in recent years. Meanwhile, these tools basically cover all the code representation forms and mainstream detection techniques.

These tools are categorized into two mainstream clone detection approaches, i.e., traditional clone detection, and deep learning-based clone detection. From the view of code representation, there are a variety of features extracted from code, such as text by SDD, token by CCFinder, SourcererCC, Oreo, CCAligner and CCLearner, AST by Deckard and ASTNN, and other semantic representation by DeepSim, spanning from text, token, and syntax to semantics; from the clone detection algorithm or model employed, for DL-based tools, ASTNN, DeepSim, CCLearner focus on abstracting deeper semantic information through various deep learning models, i.e., ASTNN by *bidirectional RNN* (Schuster and Paliwal 1997), DeepSim by *Multilayer Perceptron* (Gardner and Dorling 1998) and CCFinder by *DNNs* (Szegedy et al. 2013); for traditional tools, Deckard,

**Table 2** Code clone detectors and their manifest including extracted feature, used dataset, and supported clone types

| Tools | Venue | Method | Feature | Dataset | Clone Type | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | T1 | T2 | ST3 | MT3 | T4 |
| CCFinder (Kamiya et al. 2002) | TSE 2002 | token normalization + token-wised comparison | Token | JDK 1.3.0, FreeBSD 4.0 | ✓ | ✓ | | | |
| SDD (Lee and Jeong 2005) | OOPSLA 2005 | inverted index + N-neighbor | Text | JDK 1.5, httpd-2.0.54 | ✓ | ✓ | ✓ | | |
| Deckard (Jiang et al. 2007) | ICSE 2007 | Locality Sensitive Hash | AST | JDK 1.4.2, Linux kernel 2.6.16 | ✓ | ✓ | ✓ | | |
| SourcererCC (Sajnani et al. 2016) | ICSE 2016 | Filtering Heuristics | Token | BigCloneBench, Mutation/Injection | ✓ | ✓ | ✓ | | |
| Oreo (Saini et al. 2018) | ESEC/FSE 2018 | action token + metric comparison | Token | BigCloneBench | ✓ | ✓ | ✓ | | |
| CCAligner (Wang et al. 2018) | ICSE 2018 | code window + edit distance | Token | JDK 1.2.2, OpenNLP 1.8.1 | ✓ | ✓ | ✓ | | |
| DeepSim (Zhao and Huang 2018) | FSE 2018 | Multilayer Perceptron | CFG | BigCloneBench, GCJ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ASTNN (Zhang et al. 2019) | ICSE 2019 | Bidirectional RNN | AST | BigCloneBench | ✓ | ✓ | ✓ | ✓ | ✓ |
| CCLearner (Li et al. 2017) | ICSME 2017 | DNNs | Token | BigCloneBench | ✓ | ✓ | ✓ | ✓ | |

SourcererCC, Oreo focus on improving detection efficiency by utilizing *Locality Sensitive Hash (LSH)* (Datar et al. 2004), *Filtering Heuristics* (Sajnani et al. 2013) and *Action Filtering* respectively, and CCFinder, CCAligner on the abstraction of code information through symbolic processing and code window. The workflow of the approaches adopted by the nine clone detectors are listed as follows.

- **CCFinder** First, code snippets are converted into token sequences, followed by symbolic processing and encoding, thereby computing the similarity between clone pairs.
- **SDD** It first converts the source code into code block sequence, and builds an inverted index for it, then uses the n-nearest neighbor algorithm to calculate the similarity between the source code snippets.
- **Deckard** The source code is firstly converted into an abstract syntax tree and traversed in preorder. Each of the subtrees is represented as one vector which then hashed through the LSH. The similarity is calculated based on it.
- **SourcereCC** It converts the source code into a code block sequence represented by tokens and the Filtering Heuristics has been adopted to construct an inverted index for the blocks and reduce its size. Finally, the similarity is measured based on the similar blocks matched between the code snippets.
- **Oreo** The code snippets are transformed into token sequences and then filtered through size and Action Filtering to obtain the candidate clone pairs, which will be finally judged based on the metrics comparison.
- **CCAligner** It first converts code snippets into tokens and symbolizes them, which are then divided by the size of the window predefined. The windows containing similar codes are screened out as candidates, which will be finally judged through the similarity measurement function.
- **CCLearner** The code snippets are firstly converted into token sequences and then embedded through word2vec, which put forward to model DNNs for similarity comparison between code pairs.
- **DeepSim** It first constructs control flow and data flow based on the source code snippet, then encodes them into one semantic matrix, which is sent to one multilayer perception model for clone code snippets detection.
- **ASTNN** It first transforms the source code into an abstract syntax tree and divides it into several subtrees based on one set division rule. The word2vec is utilized to embed the subtrees into vectors, which are then sent to model bidirectional RNN for clone detection training and judgment. It is noted that different from the detectors introduced above, it could give out the type of clone pair instead of a simple judgment about if it is a clone or not.

This framework can be used for the performance comparison between these two types of approaches and easily extended to integrate other clone detection tools to be assessed.

## Evaluation

In this section, we first introduce the research questions to answer and how we set up our experiments and then evaluate the efficacy of our evaluation framework.

We intend to answer the following questions.

**RQ1.** How are clone detectors affected by obfuscation, spanning from obfuscation strategies to tools?

**RQ2.** What are the detection differences between traditional and deep learning-based clone detectors in front of obfuscation strategies?

### Experiment setup

#### Clone dataset

We collect 7,936 code pairs from BigCloneBench, which consist of 6,512 clone pairs and 1,424 non-clone pairs, i.e., $|\{\langle m_1, m_2, t\rangle\}| = 7,936$ as shown below.

| T1 | T2 | ST3 | MT3 | T4 | $\perp$ |
|---|---|---|---|---|---|
| 2,084 | 1,125 | 1,450 | 1,062 | 701 | 1,424 |

For each clone or non-clone pair $\langle m_1, m_2\rangle$, we perform obfuscation on $m_2$ with 69 strategies as discussed in Sect. 3.2 and generate 282,845 obfuscated methods, termed as $m_2'$. These methods, together with $m_1$, can form 524,148 code pairs, among which clone pairs amount to 428,695, and non-clone pairs, i.e., $\langle m_1, m_2', \perp\rangle$, come to 95,453. More details about the dataset can be viewed at Anonymous (2022). It is noted that we did not evaluate clone detectors with the pairs $\langle m_1', m_2'\rangle$ ($m_1'$ is the obfuscated version of $m_1$) since this cannot reveal the effects of obfuscation.

**Experimental configurations** In our experiment, we evaluate a total of 9 clone detectors, including 5 traditional clone detectors and 4 deep learning-based ones. Most of the experiments are conducted in a server running Ubuntu with Intel(R) Xeon(R) CPU E5-2620 @ 2.10GHz. Moreover, we set up three virtual machines that are running Windows XP for SDD, Windows 10 for

**Table 3** Performance of clone detectors on not-yet-obfuscated clones as per type

| | T1 | T2 | ST3 | MT3 | T4 |
|---|---|---|---|---|---|
| SDD | **(100, 21, 34)** | **(100, 32, 49)** | **(100, 21, 34)** | **(100, 2, 3)** | **(100, 0, 0)** |
| CCFinder | (100, 100, 100) | (98, 100, 99) | **(100, 38, 55)** | **(97, 3, 7)** | **(75, 0, 1)** |
| CCAligner | (92, 99, 95) | (86, 97, 91) | (87, 81, 84) | **(73, 48, 58)** | **(54, 31, 39)** |
| Deckard | (99, 100, 100) | (82, 100, 90) | (99, 80, 88) | **(96, 26, 40)** | **(71, 4, 8)** |
| SourcererCC | **(100, 43, 60)** | **(100, 5, 9)** | **(100, 3, 6)** | **(100, 1, 2)** | **(100, 0, 0)** |
| Oreo | (100, 66, 79) | **(99, 22, 35)** | **(100, 35, 52)** | **(99, 6, 11)** | **(60, 0, 0)** |
| ASTNN | (100, 100, 100) | (100, 92, 96) | (99, 95, 97) | (100, 92, 96) | (100, 89, 94) |
| DeepSim | (87, 100, 93) | (65, 100, 76) | (76, 66, 70) | (61, 50, 54) | **(31, 25, 28)** |
| CCLearner | (97, 100, 99) | (95, 100, 97) | (96, 95, 95) | (93, 71, 81) | **(61, 13, 22)** |

CCFinder, and Ubuntu 10 for SourcererCC. For traditional clone detectors, we have optimized their parameters as [61] for better analysis. For deep learning-based approaches, we follow the network structures as their papers and train the models with clone pairs from BigCloneBench. In particular, we uniformly utilize the same dataset containing 58,521 clone pairs collected from BigCloneBench for training ASTNN, DeepSim, and CCLearner.

### Efficacy

With the same testing clone pairs, we first evaluate the accuracy of clone detection among the deployed tools. We feed all the original clone pairs before obfuscation to the nine clone detectors and calculate the precision, recall, and F1 score, respectively. Table 3 shows the detection performance of the nine clone detectors for the five types of clone pairs before obfuscation. Each cell in this table contains a triad, the elements of which denote *Precision*, *Recall*, *F1-Score* in turn. Generally, a clone detector is regarded as being effective if its clone detection rate is above 50%. It is observed that except for some traditional tools, other tools perform effectively on two to five clone types. In particular, the detection rates of T1 and T2 clone pairs by various detectors are all close to 100%. Most of the detectors can achieve a higher detection rate than 80% on ST3 clones.

As for obfuscators, we conduct an experiment to measure their practicality by testing whether the obfuscated code can be executed as expected. Here we utilize Google Code Jam Google (2020) and randomly select 10 programs from each of the 30 programming problems belonging to all the six rounds of contests held in one year and obtain 300 ones in total, which covers a considerable number of mainstream algorithms used in the real scenarios. We obfuscate these Java projects via six obfuscators selected in our work with the corresponding obfuscation strategies, which include all the basic strategies of each tool and combination strategies shown to be representative, totally 34 ones. The result shows that 99% of 10,132 obfuscated projects can be run normally. The details of the strategies and the percentage of runnable projects after obfuscation could be referred to [61]. Through a manual investigation, we obtain three main reasons that lead to a failure of obfuscation: (1) *JDK incompatibility*. Many exceptions "`java.lang.IllegalAccessError`" are caught in JBCO-obfuscated code since JBCO adds a lot of accesses to the classes in the package `jdk.*`. However, the latest JDK undergoes significant changes where the previous classes are no longer available. (2) *Object conflicts.* Junk variables may be added by obfuscators that can lead to a conflict if there exist variables with identical names. It is found in NCO-obfuscated code where a member field with the same name is added into the class and it causes a compilation error. (3) *Bytecode breakdown.* The bytecode may be destroyed by an obfuscator. For example, Radon may empty the main method during obfuscation which causes a runtime error. Several programs fail to run after obfuscation by ProGuard because the name of the entrance class and method are replaced when using *IR* strategy. A number of programs fail to execute after being obfuscated with the strategy *CFO*. The error is thrown with JNI errors, for which the reason is probably the destruction of the Java bytecode during the process of using *CFO* strategy.

## Data analysis and measurement

In this section, we conduct a comprehensive analysis in terms of our research questions. Due to the space limitation, we cannot present all experimental results and analysis in the paper, but interested readers can visit [61] for more details. It is noted that for consistency, we uniformly use ***Recall*** to measure the detection rate of clone detectors for both the true and false clone pairs. In addition, for the assessment of obfuscation strategies, as the same obfuscation of different obfuscators could be different, we uniformly use the strategies of ***Radon*** as the evaluation target.

On one hand, ***Radon*** covers all the four types of obfuscation strategies as shown in Table 2; on the other hand, the overall impact on the detection rate of clone detectors is more obvious, thus bringing a better presentation effect.

## Clone detectors under obfuscation

In this section, we aim to investigate the influence of obfuscation in clone detection by comparing the detection rate between original code and obfuscated code. More specifically, it consists the influence of both *obfuscation strategies and tools*.

### *Changes in the detection rate of clone detectors before and after code obfuscation*

After obfuscating the code using obfuscation strategies, the detection rate of the clone code by the clone detectors has been significantly affected except when ASTNN is coping with ST3, MT3, T4 clones. We have made statistics on the changes in the detection rate of various types of clones by the clone detector after obfuscation as shown in Fig. 3. In general, the detection rate of the clone detector falls in the range of 21%-100%. The tool with the biggest drop is Deckard, whose detection rate of the corresponding clone type drops by between 80% and 100%. For clone types that the clone detectors cannot detect effectively, as the bold font in Table 3 shows, after being affected by the obfuscation strategy, the ability to detect the corresponding clone code type is basically lost.

*Finding 1* Obfuscation has a great impact on the detection rate of clone detectors. Except for some special cases, the detection rate of these detectors on varying types of clones can be reduced by an average of 21%-100%, and hence they cease to work towards the obfuscated code.

### *Influence of basic obfuscation strategies*

In order to better measure the impact of obfuscation strategies, we compare the impact of the basic obfuscation strategies on the clone detectors and find that the degree of its impact is not necessarily proportional to the *level of obfuscation*. For example, some seemingly simple obfuscation strategies, such as *IR* and *NCO*, have a greater impact on the detection results. Table 4 records the average impact brought by the obfuscation strategies of Radon on the detection rate of different clone types. The symbol "*" indicates an unavailable result since the detection rate of SDD and SoucererCC on clone pairs are all below 50%, and ASTNN could not analyze the test records obfuscated by several strategies of Radon because of its own design problems. As to distinguish the traditional tools and deep-learning based tools, we calculate the average impact on them respectively. The result is represented as $(x_1, x_2)$, where $x_1$ represents the impact on

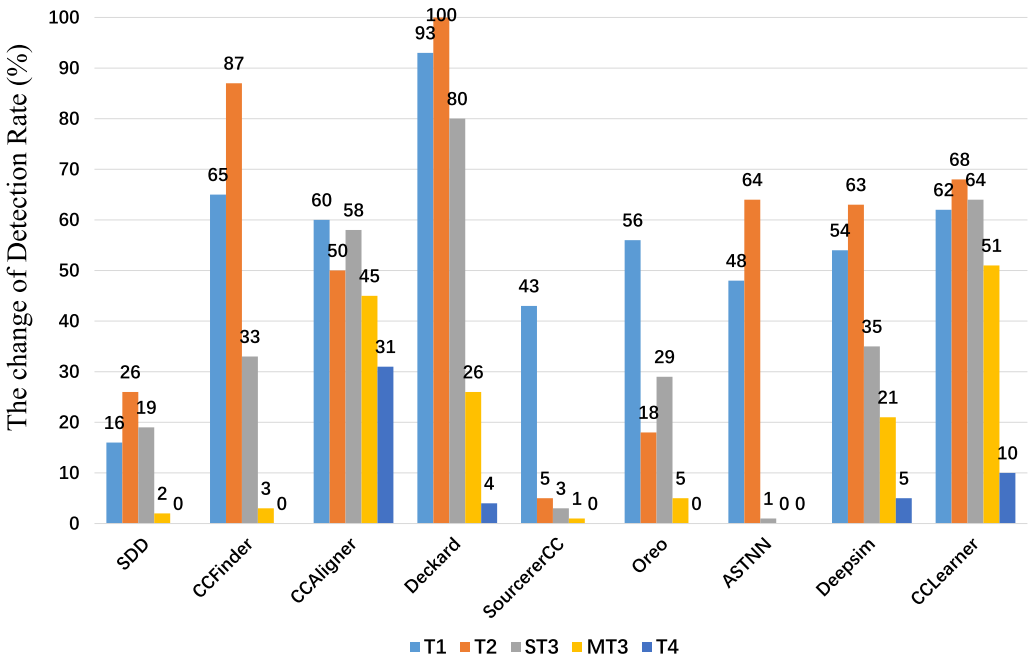


**Fig. 3** Average detection rate change of clone detectors on T1, T2, ST3, MT3, T4 under the impact of the various strategies of Radon

**Table 4** The change of the detection rate of the clone detectors under the impact of strategies of Radon

| Tools | Basic Strategy (%) | | | | | 2-Combined (%) | | | | | | | | | | 3-Combined (%) | | | | 4-Combined (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | IR | NCO | SCE | ISC | CFO | IR_NCO | IR_SCE | IR_ISC | IR_CFO | NCO_SCE | NCO_ISC | NCO_CFO | SCE_ISC | SCE_CFO | ISC_CFO | IR_NCO_ISC | IR_NCO_CFO | IR_ISC_CFO | NCO_ISC_CFO | IR_NCO_ISC_CFO |
| SDD | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| CCFinder | 60 | 60 | 49 | 2 | 100 | 74 | 75 | 50 | 100 | 100 | 49 | 100 | 31 | 100 | 100 | 74 | 100 | 100 | 100 | 100 |
| CCAligner | 11 | 36 | 19 | 10 | 93 | 38 | 14 | 11 | 93 | 55 | 36 | 93 | 13 | 93 | 93 | 38 | 93 | 93 | 93 | 93 |
| Deckard | 79 | 93 | 93 | 77 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 | 93 |
| SourcererCC | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| Oreo | 66 | 64 | 1 | 0 | 66 | 66 | 66 | 66 | 66 | 65 | 64 | 66 | 1 | 66 | 66 | 66 | 66 | 66 | 66 | 66 |
| **Average** | 51 | 61 | 43 | 31 | 80 | 66 | 60 | 49 | 80 | 76 | 60 | 80 | 42 | 80 | 80 | 66 | 80 | 80 | 80 | 80 |
| ASTNN | 29 | 24 | 18 | 20 | 15 | 33 | 28 | 29 | 14 | 23 | 27 | * | 20 | 20 | 17 | 35 | * | 12 | * | * |
| DeepSim | 28 | 45 | 6 | 0 | 49 | 65 | 43 | 37 | 42 | 54 | 48 | 68 | 3 | 45 | 35 | 62 | 39 | 63 | 64 | 65 |
| CCLearner | 68 | 7 | 0 | 0 | 92 | 89 | 68 | 68 | 92 | 6 | 8 | 92 | 0 | 92 | 92 | 89 | 92 | 92 | 92 | 92 |
| **Average** | 42 | 25 | 8 | 7 | 52 | 62 | 46 | 45 | 49 | 28 | 28 | ** | 8 | 52 | 48 | 62 | ** | 56 | ** | ** |

traditional tools and $x_2$ is the impact on deep learning-based ones.

Averagely, the impact of *IR* on traditional tools and deep learning tools is (51%, 42%), that of *NCO* is (61%, 25%), surpassing the structure strategy *ISC* whose impact is only (30%, 7%). As for *CFO*, it induces the largest detection rate drop for both traditional (95%) and deep learning-based tools (52%).

*Finding 2* Even simple obfuscation strategies (e.g., IR and NCO) can have a serious impact on the detection rate of clone detectors, compared with more complex strategies (e.g., *ISC, CFO*).

**Cause analysis.** We manually examine the obfuscated code and explain this phenomenon with regard to the code representation methods considered by clone detectors.

Firstly, it should be clear that simple strategies (e.g., *IR*) can either replace or encode the identifiers or strings in code, which significantly changes the textual information; *CFO* inserts a large number of branch statements like "`switch`" and "`goto`", then affects both textual and structural information greatly; structure obfuscation adjusts the order of specific statements or methods, possessing a certain effect on structural information but little on textual one.

- *Token based detectors–CCFinder, CCAligner, Oreo, and CCLearner*. Simple strategies like *IR* have almost the same effect as *CFO*, far exceeding structure obfuscation (e.g., *ISC*). As token mainly reflects textual information of the code snippet.
- *AST based detectors–Deckard and ASTNN*. AST is a type of representation that reflects both the textual and structural information of the code snippet. Then, the effect of simple strategy coincide *CFO*, slightly higher than structure obfuscation.
- *CFG, DFG based detector–DeepSim*. The impact of *NCO* and *CFO* is roughly the same, which all exceed other obfuscation strategies. As the **feature matrix** of DeepSim comes from **CFG** and **DFG** of code snippets, then passed to **Multilayer Perceptron model** for further semantic mining, *NCO* could cause greater impact to **DFG**, and *CFO* to **CFG**.

**Summary.** *It could be observed that firstly, involving multiple (e.g., DeepSim) or more complex code representations (e.g., ASTNN) can indeed dig deeper into code semantics, but it also introduces more attack surfaces, as the code snippet has been depicted from more types of feature information. For example, compared with the token-based tools, AST-based ones are more susceptible to structure strategy, as the structure information could be reflected in AST but not token, which only contains textual information; Secondly, the effect of various types of obfuscation strategies on clone detector is closely related to its code representation method, from which the features have been extracted and then encoded for further similarity calculation between code pairs.*

#### *Basic obfuscation strategies vs. combined strategies*

Perhaps the most surprising thing is that the combined strategy's influence on clone detectors is not necessarily better than a single strategy, and sometimes it may be severely weakened. As shown in Table 4, the impact of *IR* on the detection rate of CCAligner was originally 11% and keeps 11% after being combined with strategy *ISC*; the impact of *IR* on the detection rate of ASTNN is originally 29% and drops to 14% after combined with *CFO*. Overall, the average influence of *IR* on the detection rate of clone detectors is (51%, 42%), while the combination strategy including *IR* has an influence on traditional clone detectors and deep learning ones in the range of [11%, 100%] and [0%, 92%] respectively.

*Finding 3* The combination strategy is not necessarily better than the original single strategy. If one strategy is combined with another of the same obfuscation level, it more likely becomes stronger. However, it does not exhibit a complementary effect between one single strategy and a more complex one. If this one's interference is bigger, the combination tends to be stronger; if not, then more likely to be the same or even weaker.

#### *False positives caused by code obfuscation*

In the previous sections, we have measured how many true clone pairs cannot be detected after obfuscations. Here we

**Table 5** The drop (%) in detection rate of false clone pairs of ASTNN and DeepSim under various obfuscators

| | Radon | Obfuscator | JBCO | JODE | yGuard | ProGuard |
|---|---|---|---|---|---|---|
| ASTNN | 4 | 3 | 3 | 9 | 9 | 6 |
| DeepSim | 0 | 0 | 3 | 4 | 0 | 0 |

investigate whether any false clone pairs, i.e., $\langle m_1, m_2, \perp \rangle$, can be falsely classified as a clone.

It is observed that under certain circumstances the false positive rate has been increased by varying percents for clone detection. This phenomenon is pervasive among these obfuscation tools, in the two deep learning-based clone detectors, ASTNN and DeepSim as shown in Fig. 5. Taking ASTNN as an example, it is affected by almost all the obfuscation tools and their corresponding strategies.

We further study the situation of *yGuard*, which causes a 9% drop using the strategy *IR*, with the number of being falsely marked as clone pairs rising from 6 to 130 averagely out of 1,488 pieces of records in total. Although the rise may be minor in terms of rate, the change can also be obvious as the number of programs to be analyzed increases. Therefore, in reality, this situation can be really serious when detecting massive of data and becomes unacceptable to the benign users who use obfuscation tools for legal purposes. Figure 5 shows that *Obfuscator* has the weakest influence, which only affects ASTNN with the smallest impact (3%).

*Finding 4* Almost all the obfuscators make false clone pairs after obfuscation easier to be falsely recognized by certain DL-based clone detectors, producing more false positives, possibly because some false clone pairs after obfuscation have exceeded their learning range. Moreover, Obfuscator is the weakest one that may cause these alarms, which may be one ideal obfuscation tool for benign developers to protect their own code while avoiding being misjudged.

**Cause analysis.** We adopt a mask-based approach to analyze which statements in the obfuscated code fragment affect the judgement. As shown in Fig. 4, for ASTNN, $\langle a, c \rangle$ is originally judged as a false clone, after obfuscation, $\langle b, c \rangle$ judged as a clone. For each statement in $b$, we comment it out, then $\langle b, c \rangle$ fed to ASTNN for judgement. Further, for the conditional expression in the If statement, we modify it to False to verify its influence. If the result is true, it means this statement is not a key factor. Otherwise, it is that. Lastly, the *If statement* is judged as the key factor, which is inserted by the obfuscator.

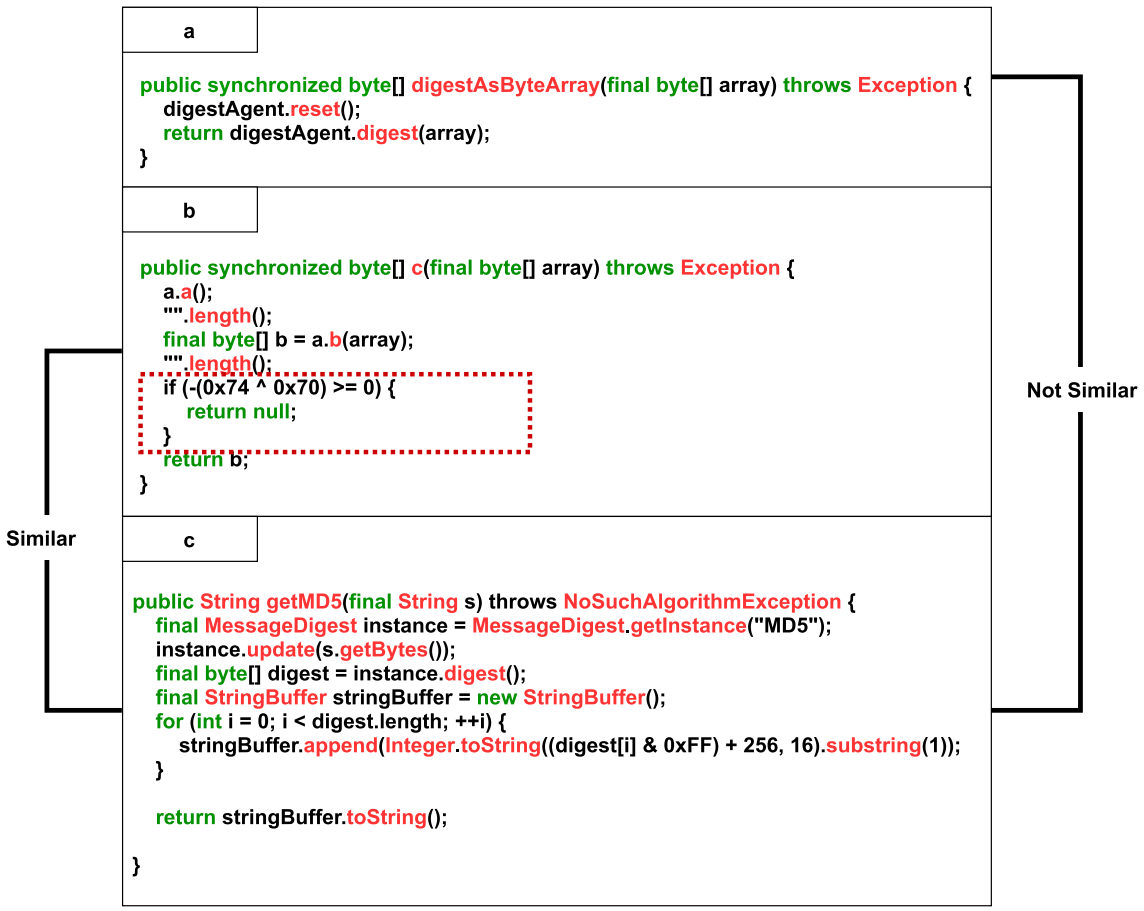


**Fig. 4** An example of false clone pair misjudged after obfuscation

Meanwhile, taking *Radon*, *Obfuscator*, *JBCO* which all have various obfuscation strategies for comparison, the effect of *Radon* and *JBCO* all exceed *Obfuscator*. As *Radon* may insert lots of lines of code, *JBCO* may convert the identifiers or constants to a more complicated form, but *Obfuscator* mainly makes some simpler replacements or modifications.

**Summary.** *In the process of obfuscation, obfuscators could add some irrelevant statements, which may have similarities with the code snippet to be compared, thus making one original false clone pair judged as true.*

### Traditional clone detectors versus DL-based detectors

Here we compare the traditional and deep learning-based clone detectors from four aspects.

#### *Overall performance comparison*

It is observed that the detection stability of DL-based tools on the detection of clone pairs in the face of obfuscation attacks is better than that of traditional tools. As shown in Table 4, the average influence of obfuscation strategies on the detection rate of traditional tools is between 31% and 80%, and the impact on deep learning tools is between 7% and 62%.

However, the false alarm rate is just the opposite. As aforementioned, there is a minor rise in the false alarm rate of the deep learning-based tools after obfuscation, but this phenomenon never occurs for traditional tools, whereas all decrease.

*Finding 5* In general, the clone code detection rate of deep learning tools is more robust than the traditional ones. However, the situation of the false alarm rate is just the opposite.

**Cause analysis.** Compared with traditional tools, deep learning tools could conduct deeper mining and abstraction of code semantics. Therefore, in general, deep learning tools are relatively robust. However, when detecting the false clone pairs after obfuscation, the detection principle of traditional tools is to compare the difference value between the code pairs with the threshold set, as the difference value after obfuscation is almost impossible to get smaller, but basically, it may only become larger or unchanged. So the false alarm rate will not increase, but a small decrease will occur, thus showing better robustness.

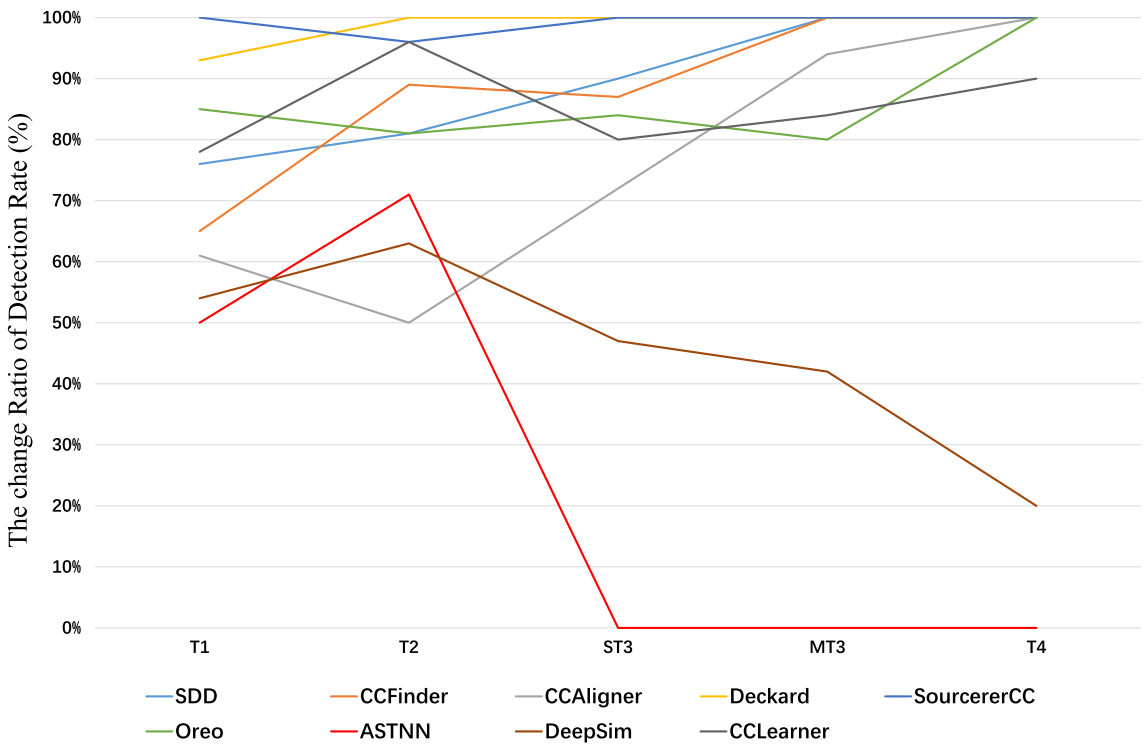


**Fig. 5** The average change ratio of detection rate of both the traditional clone detectors and the DL-based ones for various clone types under the influence of obfuscation strategies

#### *Performance comparison for various clone types*

For the accuracy and rationality of the experiment, the average change ratio of the clone detector's detection rate on T1, T2, ST3, MT3, T4 is calculated to indicate the degree of influence of obfuscation strategy on it as shown in Fig. 5. It indicates that the effect of obfuscation strategies on traditional tools has an upward trend in the detection rate of the five clone types. That is, their detection rate of simple clone types (i.e., T1 and T2) is affected relatively small, while the one of complex clone types (i.e., ST3, MT3, and T4) is affected comparatively large. However, the performance of DL-based tools is just the opposite. Especially, as shown in Fig. 5, the detection rate on ST3, MT3, T4 of ASTNN almost suffers no impact.

*Finding 6* Overall, Deep learning-based approaches exhibit a more superior resilience to obfuscation for complex clone pairs, i.e., ST3, MT3, and T4 than simple ones, e.g., T1, T2, while the performance of traditional ones is just the opposite.

**Cause analysis.** This is caused by the essential difference between traditional and DL-based tools. For traditional tools, the feature extraction method and comparison threshold are fixed. Compared with the clone pairs of ST3, MT3, and T4, the similarity between the code snippets of T1 and T2 is relatively higher. Therefore, when the same obfuscation strategy is used to disturb them, the difference between T1 and T2 clone codes is still small. Hence the probability of exceeding the similarity threshold should be small, and the degree of influence on its detection rate is corresponding weaker.

For deep learning tools, the structure and parameters of the feature extraction and comparison model are all closely related to the training set data. When the same obfuscation strategy is used to interfere with the clone code, the clone codes of ST3, MT3, and T4 are less changed compared to T1 and T2. So their detection rate is relatively less affected. According to the way the model is built, DL-based tools can be divided into two categories: first, **five two-classifier models** (e.g., ASTNN) are trained based on T1, T2, ST3, MT3, T4 clone pairs respectively. Second, **one two-classifier model** (e.g., DeepSim) is trained based on these five types of clone pairs. For the first type, the **five two-classifier models** are more suitable for the detection of their corresponding clone type. Compared with the **two-classifier model** of T1 and T2, the ones of ST3, MT3, and T4 are not so sensitive to the difference between clone pairs. For example, the abstraction of feature extraction may be higher, and the threshold setting for similarity comparison could be looser. Therefore, when using the same obfuscation strategy to disturb the clone pairs, the detection rate of ST3, MT3, and T4 will be less affected. For the second type, whose training data is composed of various types of clone data, its structure and parameters are affected by multiple clone types, then the difference between the effect on the detection rate of ST3, MT3, T4 and T1, T2 is not as large as the first type.

#### *The effect of obfuscation strategy on two types of clone detectors*

For traditional tools, among the basic obfuscation strategies, the one that has the most significance is *CFO*. Table 4 shows *CFO* makes the detection rate of all the traditional clone detectors drop by 66∼100%. From Fig. 3, we observe that the detection rate of clone types that cannot be well detected by traditional clone detectors (i.e., the detection rate is less than 50%) has almost dropped to 0%. All in all, *CFO* basically makes traditional tools lose their ability to detect clone code. All the combined strategies formed by *CFO* have exactly the same effect. The average effect of other obfuscation strategies is between 31% and 61%, which has an obvious gap.

However, the most effective basic strategy for deep learning-based tools could be *IR*, *NCO*, whose effect is up to 29%, 45% on ASTNN, DeepSim respectively, as shown in Table 4. Additionally, the combined strategy that performs most effectively is *IR_NCO*, which causes a 62% drop, much larger than others. In contrast, *CFO* has a trivial impact on some deep learning tools. For example, except for the clone pairs of T2, ASTNN's detection rate is only affected by 4% average by *CFO*.

*Finding 7* For traditional tools, the most influential strategy is control flow obfuscation. However, for deep learning tools, the most influential strategy could be some simple ones like IR and NCO.

**Cause analysis.** Similarly, we make the inference for the potential reason in the light of the different changes made by obfuscations to the features considered by clone detectors.

- For traditional tools, *CFO* can increase the difference between code snippets to a greater extent compared to other obfuscation methods, so that it is easier to exceed the preset judgment threshold and have a greater impact on the detection result.
- For deep learning tools, here, we select a more obvious example for analysis. In the experiments on ASTNN, the attack effect of *IR* is almost twice that of *CFO* (29% and 15%, respectively). The reason is that ASTNN uses word2vec to generate the initial feature vector based on the identifier in the code, and

then generates the feature vector of one code snippet according to the corresponding conversion algorithm, which will make the text information such as the identifier has a greater effect on the analysis result of the detectors. Meanwhile, the use of the *bifirectional RNN* strengthens the degree of mining corresponding semantic information and further strengthens its influence on the detection result. When the *IR* is used for obfuscation, the identifier information is greatly changed, as the ASTNN does not uniformly process the identifier, thus it will have a greater impact on the detection rate.

**Summary.** *DL-based detectors are prone to be affected by some simple strategies, as they possess a greater impact on the information which is the source of input vectors of the deep learning model and the deep learning network will perform in-depth semantic mining, so a little change in such information will have a greater impact on the judgment result.*

#### *Comparison of the impact brought by different obfuscation tools on traditional tools and DL-based ones*

Obfuscation tools perform different impacts on traditional tools and deep learning-based tools. The average effect of *JBCO* on traditional tools is 73%, which is the biggest among these tools. However, in the meantime, its effect on deep learning-based detectors is only 28%, which is also the smallest one among them. In contrast, apart from the three tools possessing only one strategy *IR*, the effect of *Radon* on traditional tools is 66%, which is a moderate effect, however, its effect on deep-learning based tools is highest among them.

*Finding 8* Among the obfuscation tools evaluated, JBCO has the greatest impact on traditional clone detectors and the least on deep learning clone detectors, which is almost opposite for Radon.

**Cause analysis.** By comparing and analyzing the code snippets after obfuscation of *JBCO* and *Radon*, it can be found that JBCO modifies code identifiers, constants, and other information to a greater extent than Radon, which only uses fixed value replacement for some variables, then on textual information; however, for structure information, the impact of Radon is bigger, as it could increase the number of code lines to tens of times and make its control flow much complicated.

As mentioned in Finding 2, most of the traditional tools evaluated in this study are based on tokens, which mainly reflect textual information of the code, then *JBCO* will have a greater impact on these tools.

For DL-based detectors, the detection rate largely depends on its training data BigCloneBench, which is composed of five types of clones, having greater tolerance for code differences caused by simple strategy and less for *CFO*. Therefore, although *JBCO* has made major changes to the textual information, it still does not exceed the learning range of the DL-based tools, the impact is corresponding weaker; *Radon* changes more structure information, exceeding the feature boundary of the deep learning model, then resulting in greater effect.

## Discussion

**Threats to validity.** First, few mature clone detectors could be directly used for assessment. Compared to traditional clone detection, deep learning-based clone detection is just a newly emerging technique, and few of them have been open-sourced. Thus, the number of clone detectors based on deep learning in our assessment is relatively small, which inevitably makes the findings have certain limitations. Second, the decompilation may cause minor changes on the original method besides the obfuscation that make the results less accurate. To mitigate it, we turn to using the decompilation version of clone methods during the whole assessment experiment.

**Selection of obfuscators.** In this study, we choose six commercial code obfuscators to transform Java bytecode while not using source code obfuscators is threefold: (1) it has a wide applicable scenario in reality, especially for malicious code detection, vulnerability hunting, and code copyright infringement detection. For example, Android is now the biggest platform for Java projects, and most Android apps are compiled into bytecode before distribution. Many works have proposed their own methods to detect clones from the bytecode of Android apps (Crussell et al. 2012; Wang et al. 2015, 2020). (2) to our best knowledge, there are currently no public obfuscators for Java source code. Although a previous study (Schulze and Meyer 2013) has implemented a semi-automated code obfuscator, its functionality and robustness cannot rival these commercial ones. Additionally, since the Java code in BigCloneBench is not complete, for example, with many missing declarations for variables and methods. Source code obfuscators may not be able to perform advanced strategies like control flow obfuscations to the code. (3) Last, to minimize the noise caused by compilation and decompilation, we only use the decompiled version of these Java methods. Meanwhile, the clone type almost (92%) keeps unchanged after decompilation according to our observation.

**Improvement suggestions.** Based on the findings and analysis, we summarize a number of suggestions for users from different perspectives.

***From the perspective of designers of clone detectors***

- Considering the attack surface, it's more effective and safer for the defenders to choose a simpler code representation and focus on optimizing the comparison algorithms or models as Finding 2. That is why CCAligner performs well in both the detection rate and the anti-interference ability. For on one hand, it represents source code as **token sequence**, thus reducing the attack surfaces probably brought compared to AST or more complex code representation methods; on the other hand, one **sliding code window** is introduced, on which granularity the code pairs would be compared, capturing local sequence features, improving detection accuracy while ensuring robustness.
- It is suggested to construct one clone detector for each type of clone pair. When detecting the target, use these five detectors to analyze it one by one. As long as one of them judges it to be true, then it is one clone pair, which could not only improve the clone detection rate but also enhance the robustness of the detection model according to Finding 6.
- For deep learning-based clone detectors, we must fulfill the abstraction process of code closely related to the initialization of feature vectors as Finding 7, e.g., normalizing the identifiers of code snippet, avoiding seemingly slighter interference caused by some simple strategies.
- Except for the possible attack surface brought by code representation, developers should also be aware of the tolerance range of feature extraction and comparison algorithms with regard to the difference between code snippets from Finding 8. For traditional clone detectors, the key is to obtain more code information of higher-level abstraction, then the tolerance range is larger, thereby improving its robustness. Taking the tools based on tokens as an example, compared to CCFinder, which directly compares the similarity between code snippets based on the **token sequences**, CCAligner achieves better robustness by employing a **sliding window mechanism** to harvest more windows of code for feature extraction and similarity comparison. For DL-based ones, their tolerance for code differences largely depend on the richness of their training set, then it is recommended to select clone pairs processed by popular obfuscation tools, e.g., the clone pairs obfuscated in this study, to supplement their training set data, so as to have better resistance.

***From the perspective of analyzers who are eager to evaluate clone detectors***

- According to Finding 2, it is suggested that the code representation method of clone detector could be inferred by comparing the effects of different obfuscation strategies, e.g., if the effects of simple strategy and *CFO* are almost the same, far greater than structure ones, then the method should be taken.
- For a black box clone detection model, users can determine the specific type of the model by observing and comparing the changes of detection rate of different clone types. For example, according to the impact of T1, T2 and ST3, MT3, T4, it can be judged to be either a traditional model or a deep learning one. Based on the magnitude of the difference between them, it could be further judged as the first or second type of DL-based tools from Finding 6.
- From Finding 7, users can first use a variety of single obfuscation strategies to test the deep learning tools. If some simple strategies have a more prominent impact on the clone detector than other strategies, the corresponding code information interfered by the strategy can be used as the adjustment target for the generation of adversarial samples. For instance, the identifier information could be the target for ASTNN. In this way, on one hand, since only the layout information of the code needs to be adjusted, the difficulty of generating adversarial samples can be greatly reduced, and at the same time, the adversarial attack could achieve the best effect.

***From the perspective of users and designers of obfuscators***

- It is recommended that the designers of obfuscators could follow the concept of *Obfuscator*, do more replacement or modification, and try to avoid insertion operations as *Radono* or *JBCO*. In this way, on one hand, as depicted in Finding 4, reducing the false alarm rate for benign users who just want to protect their own codes; on the other hand, diminishing the possible damage brought by attackers utilizing relatively powerful obfuscators, i.e., *Radon* and *IBCO* shown in Finding 8. Meanwhile, this could also help improve the efficiency of obfuscators as fewer process needed. During our evaluation, the process of *Radon* or *JBCO* could be hours of time, but *obfuscator* only in minutes, which is ideal for the users.

## Conclusion

In order to evaluate the effect of code obfuscation in clone detection, we build an evaluation framework integrating six commercial obfuscation tools and nine clone detectors. In particular, there are six traditional clone detectors and three deep learning-based detectors as evaluation subjects. We collect a number of Java code pairs from BigCloneBench and perform 69 strategies to obfuscate them. Last, we obtain 428,695 true clone pairs and 95,453 false pairs which are fed into clone detectors for evaluation. Two analyses are subsequently conducted to measure the effect of varying obfuscation strategies and tools and the different performance of traditional and deep learning-based clone detection. Eight findings and discussion have revealed the issues in both code obfuscation and clone detection as well as improvement suggestions.

**Acknowledgements**
We would like to thank the anonymous reviewers for detailed comments and useful feedback.

**Authors' Information**
Guozhu Meng obtained his Ph.D degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore at 2017. His supervisors are Full Prof. Liu Yang and Full Prof. Zhang Jie. He joined Institute of Information Engineering of Chinese Academy of Sciences as Associate Professor in 2018. His research focuses on system security and artificial intelligence security as follows: Android security, big data analysis, vulnerability detection, and AI security and privacy.

**Author Contributions**
WH: investigation, conceptualization, methodology, materials, writing, editing, experiment, validation, review, resources. GM: methodology, discussion, writing, review, supervision. CL: investigation, resources, discussion, experiment, validation, review. QY: resources, discussion, experiment, review. KC: discussion, review, supervision. ZM: discussion, review, supervision. All authors read and approved the final manuscript.

**Funding**
IIE authors are supported in part by the National Key R &D Program of China (2020AAA0140001), NSFC U1836211, Beijing Natural Science Foundation (No. M22004), the Anhui Department of Science and Technology under Grant 202103a05020009, Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI) and a research grant from Huawei.

**Availibility of data and materials**
We will publish our benchmark and evaluation framework after the work has been accepted.

## Declarations

**Competing interests**
The authors declare that they have no competing interests

Received: 22 December 2022 Accepted: 22 February 2023
Published online: 02 July 2023

**References**

Ain QU, Butt WH, Anwar MW, Azam F, Maqbool B (2019) A systematic review on code clone detection. IEEE Access 7:86121–86144
Anonymous: CloneVsObf. https://github.com/CloneVsObf/CloneVsObf (2021)
Anonymous: Impacts of obfuscation on clone detection. https://sites.google.com/view/obf-clone-eval/ (2022)
Apache: The Apache Ant Project. https://ant.apache.org/ (2020)
Balakrishnan A, Schulze C (2005) Code obfuscation literature survey. CS701 Construction of compilers, 19
Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591
Ben-Nun T, Jakobovits AS, Hoefler T (2018) Neural code comprehension: a learnable representation of code semantics. In: Advances in neural information processing systems, pp 3585–3597
Cao L, Sun G, Wang H, WANG S (2006) Logic invariability study of junk code transformation. Computer Engineering 20, 048
Chen J, Alalfi MH, Dean TR, Zou Y (2015) Detecting android malware using clone detection. J Comput Sci Technol 30(5):942–956
Cimato S, De Santis A, Petrillo UF (2005) Overcoming the obfuscation of java programs by identifier renaming. J Syst Softw 78(1):60–72
Cimitile A, Martinelli F, Mercaldo F, Nardone V, Santone A (2017) Formal methods meet mobile code obfuscation identification of code reordering technique. In: 2017 IEEE 26th International conference on enabling technologies: infrastructure for collaborative enterprises (WETICE), pp 263–268. IEEE
clone: overview of clone detection tools for java. https://github.com/c-oberle/clone-detection-tools (2020)
Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations. http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial
Crussell J, Gibler C, Chen H (2012) Attack of the clones: Detecting cloned applications on android markets. In: Computer Security—ESORICS 2012, pp 37–54. Springer, Berlin, Heidelberg
Datar M, Immorlica N, Indyk P, Mirrokni VS (2004) Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the twentieth annual symposium on computational geometry, pp 253–262
Duala-Ekoko E, Robillard MP (2007) Tracking code clones in evolving software. In: 29th international conference on software engineering (ICSE'07), pp 158–167. IEEE
Gardner MW, Dorling SR (1998) Artificial neural networks (the multilayer perceptron)-a review of applications in the atmospheric sciences. Atmos Environ 32:2627–2636
Göde N, Koschke R (2011) Frequency and risks of changes to clones. In: Proceedings of the 33rd international conference on software engineering, pp 311–320
Google: Google Code Jam. https://codingcompetitions.withgoogle.com/codejam (2020)
Guardsquare: ProGuard. https://github.com/Guardsquare/proguard (2020)
Hammad M, Garcia J, Malek S (2018) A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In: Proceedings of the 40th international conference on software engineering, pp 421–431
Hoenicke J (2020) JODE. http://jode.sourceforge.net/
ItzSomebody: radon. https://github.com/ItzSomebody/Radon (2020)
JavaParser: tools for your Java code Transform. https://javaparser.org/ (2020)
Jiang L, Misherghi G, Su Z, Glondu S (2007) Deckard: Scalable and accurate tree-based detection of code clones. In: 29th international conference on software engineering (ICSE'07), 96–105
Kamiya T, Kusumoto S, Inoue K (2002) Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670
Kim S, Woo S, Lee H, Oh H (2017) Vuddy: A scalable approach for vulnerable code clone discovery. In: 2017 IEEE symposium on security and privacy (SP), pp 595–614. IEEE
Krinke J (2001) Identifying similar code with program dependence graphs. In: Proceedings eighth working conference on reverse engineering, pp 301–309. IEEE
Kuhn A, Ducasse S, Gírba T (2007) Semantic clustering: identifying topics in source code. Inf Softw Technol 49(3):230–243
Lee S, Jeong I (2005) Sdd: high performance code clone detection system for large scale source code. In: Companion to the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, pp 140–141

Li L, Feng H, Zhuang W, Meng N, Ryder B (2017) Cclearner: A deep learning-based clone detection approach. In: 2017 IEEE international conference on software maintenance and evolution (ICSME), pp 249–260. IEEE
Liu Z, Wei Q, Cao Y (2017) Vfdetect: A vulnerable code clone detection system based on vulnerability fingerprint. In: 2017 IEEE 3rd information technology and mechatronics engineering conference (ITOEC), pp 548–553. IEEE
Livieri S, Higo Y, Matsushita M, Inoue K (2007) Analysis of the linux kernel evolution using code clone coverage. In: Fourth international workshop on mining software repositories (MSR'07: ICSE Workshops 2007), pp 22–22. IEEE
Meyer D, Schulze D-IS (2012) Analyzing the robustness of clone detection tools regarding code obfuscation. Bachelor thesis, University of Magdeburg
Monden A, Nakae D, Kamiya T, Sato S-i, Matsumoto K-i (2002) Software quality analysis by code clones in industrial legacy software. In: Proceedings eighth IEEE symposium on software metrics, pp 87–94. IEEE
Nguyen HA, Nguyen TT, Pham NH, Al-Kofahi JM, Nguyen TN (2009) Accurate and efficient structural characteristic feature extraction for clone detection. In: International conference on fundamental approaches to software engineering, pp 440–455. Springer
OKane P, Sezer S, McLaughlin K (2011) Obfuscation: the hidden malware. IEEE Secur Priv 9(5):41–47
Ragkhitwetsagul C, Krinke J (2017) Using compilation/decompilation to enhance clone detection. In: 2017 IEEE 11th international workshop on software clones (IWSC), pp 1–7. IEEE
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
Roy CK, Cordy JR (2009) A mutation/injection-based automatic framework for evaluating code clone detection tools. In: 2009 International conference on software testing, verification, and validation workshops, pp 157–166. IEEE
Roy CK, Cordy JR (2007) A survey on software clone detection research. Queen's Sch Comput TR 541(115):64–68
Sable: JBCO. http://www.sable.mcgill.ca/JBCO/ (2020)
Saini V, Farmahinifarahani F, Lu Y, Baldi P, Lopes CV (2018) Oreo: Detection of clones in the twilight zone. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 354–365
Sajnani H, Saini V, Lopes CV (2013) A parallel and efficient approach to large scale clone detection. 2013 7th international workshop on software clones (IWSC), 46–52
Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV (2016) Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th international conference on software engineering, pp 1157–1168
Schulze S, Meyer D (2013) On the robustness of clone detection to code obfuscation. In: 2013 7th international workshop on software clones (IWSC), pp 62–68
Schuster M, Paliwal KK (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45:2673–2681
Sheneamer A, Kalita JK (2016) A survey of software clone detection techniques. Int J Comput Appl 137:1–21
Soot: Soot: a Java Optimization Framework. https://www.sable.mcgill.ca/soot/ (2020)
Steiger S (2020) Procyon. https://github.com/ststeiger/procyon
superblaubeere27: Obfuscator. https://github.com/superblaubeere27/obfuscator/ (2020)
Svajlenko J, Islam JF, Keivanloo I, Roy CK, Mia MM (2014) Towards a big data curated benchmark of inter-project code clones. In: 2014 IEEE international conference on software maintenance and evolution, pp 476–480. IEEE
Szegedy C, Toshev A, Erhan D (2013) Deep neural networks for object detection. Adv Neural Inf Proces Syst 26
Tufano M, Watson C, Bavota G, Di Penta M, White M, Poshyvanyk D (2018) Deep learning similarities from different representations of source code. In: 2018 IEEE/ACM 15th international conference on mining software repositories (MSR), pp 542–553. IEEE
Viticchié A, Regano L, Torchiano M, Basile C, Ceccato M, Tonella P, Tiella R (2016) Assessment of source code obfuscation techniques. In: 2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM), pp 11–20. IEEE
Wang H, Guo Y, Ma Z, Chen X (2015) Wukong: A scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 international symposium on software testing and analysis, pp. 71–82. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2771783.2771795
Wang W, Meng G, Wang H, Chen K, Ge W, Li X (2020) A 3 ident: a two-phased approach to identify the leading authors of android apps. In: 2020 IEEE international conference on software maintenance and evolution (ICSME), pp 617–628. IEEE
Wang P, Svajlenko J, Wu Y, Xu Y, Roy CK (2018) Ccaligner: a token based large-gap clone detector. In: 2018 IEEE/ACM 40th international conference on software engineering (ICSE), pp 1066–1077
Wei H, Li M (2017) Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: IJCAI, pp 3034–3040
White M, Tufano M, Vendome C, Poshyvanyk D (2016) Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM international conference on automated software engineering (ASE), pp 87–98. IEEE
Wu Y, Manabe Y, Kanda T, German DM, Inoue K (2015) A method to detect license inconsistencies in large-scale open source projects. In: 2015 IEEE/ACM 12th working conference on mining software repositories, pp 324–333. IEEE
You I, Yim K (2010) Malware obfuscation techniques: a brief survey. In: 2010 international conference on broadband, wireless computing, communication and applications, pp 297–300. IEEE
yWorks: yGuard. https://www.yworks.com/products/yguard (2020)
Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), 783–794
Zhao G, Huang J (2018) Deepsim: Deep learning code functional similarity. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. ESEC/FSE 2018, pp. 141–151. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3236024.3236068

## Publisher's Note