Bytecode Similarity Detection for Obfuscated Java Android Applications

Misheelt Munkhjargal

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

May 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Misheelt Munkhjargal**

Entitled: **Bytecode Similarity Detection for Obfuscated Java Android Applications**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Emad Shihab

_____ Examiner
Dr. Juergen Rilling

_____ Examiner
Dr. Emad Shihab

_____ Supervisor
Dr. Tse-Hsun (Peter) Chen

Approved by _____
Dr. Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2024

_____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Bytecode Similarity Detection for Obfuscated Java Android Applications

Misheelt Munkhjargal

Code similarity detection has many practical applications, such as intellectual property protection, vulnerability search, and malware detection. However, existing approaches typically focus on the source code, while many third-party libraries are released in bytecode format. Hence, developers may unknowingly use third-party libraries without knowing possible license violations or vulnerabilities. In this thesis, we introduce a deep learning approach, ByClone, to detect source code clones based on Java bytecode. We collect source-code level clone data for bytecode in 140 Android applications to conduct the experiments. We find that ByClone is effective in detecting code clones based on bytecode, with a precision and recall of 78.37 and 75.24. After obfuscating the bytecode, ByClone still has a precision and recall of 82.55 and 70.95, highlighting the potential of ByClone. Finally, we find that ByClone is not sensitive to different obfuscation options. Our study highlights the potential of clone detection based on bytecode. We also release the data for future research in this direction.

Acknowledgments

I would like to express my sincere gratitude towards my supervisor Dr. Tse-Hsun (Peter) Chen for his invaluable guidance and continuous support. His brilliance taught me how to think and approach problems in a brand new way. I am highly appreciative of this incredible opportunity.

Additionally, I am beyond grateful to my parents for their unconditional love and unrelenting support towards my education. Their hard work and dedication inspire me everyday and I would not be where I am today without their sacrifices.

Finally, I would like to thank my friends whose encouragement and support have gotten me through every step of my graduate studies. Their love and kindness have been a constant source of strength and motivation.

Thank you all for your immense contributions to this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Licensing and Vulnerability Issues in Bytecode

Third-party software is widely used by practitioners from both open-source communities and commercial companies to facilitate the process of development and maintenance. Due to privacy issues and proprietary source code, many third-party software is usually only available in obfuscated bytecode (Su, Bell, Kaiser, & Baishakhi, 2017). Using such obfuscated bytecode has the risk of unknowingly integrating third-party software that contains code with licensing agreements that developers are not aware of or contains vulnerabilities (Mlouki, Khomh, & Antoniol, 2016). A breach of license agreements may lead to serious legal issues. For example, many companies have inadvertently violated the GNU General Public License (GPL) agreement (Mlouki et al., 2016). Businesses can suffer from serious lawsuits that can amount to hefty fines. Hence, it would be beneficial for companies to detect these violations ahead of time in the development process to prevent using third-party libraries that can result in violating licensing agreements.

Code clone detection techniques can be effective in avoiding such license or vulnerability issues (Ahtiainen, Surakka, & Rahikainen, 2006). If one is able to detect *source code clones* at the *bytecode level*, third-party libraries with problematic license agreements can be prevented proactively. Similarly, third-party bytecode containing vulnerabilities can be spotted by bytecode clone

detection techniques to mitigate software quality issues when source code analytical tools do not excel. In a practical setting, bytecode similarity detection can be used as a search or scanner tool, either standalone or as an extension of a code base storage software, to detect vulnernabilities, malware, or copyright infringements.

1.1.2 Currently Available Code Similarity Detection Tools

The majority of prior studies for detecting code clones have detected similarities in source code through parsing and comparing similar patterns. Source code similarity detection is a useful technique for license and vulnerability detection (Ahtiainen et al., 2006; Ragkhitwetsagul, Krinke, & Clark, 2018; Đurić & Gasevic, 2013). However, the source code is not always accessible for code similarity analysis. For business and privacy/security reasons, the source code of the software is often not available. Instead, it is likely that the bytecode of the software is available for use. In cases when only the bytecode is accessible, our goal is to still be able to detect similarities in the software. In this thesis, we aim to advance code similarity detection techniques for the Java bytecode in Android applications.

Although some prior studies (Liu, 2021; Tang, Luo, Fu, & Zhang, 2020; Yang, Fu, Liu, Yin, & Zhou, 2021) explore source code similarity detection based on binary code, they primarily focus on using assemble code (i.e., compiled from C/C++ code). There is a lack of techniques for detecting source code similarity based on bytecode, especially for Java, where Java bytecode is peculiar due to its cross-platform nature.

1.1.3 Detecting Clones in Obfuscated Java Bytecode

A potential challenge in detecting Java bytecode in Android applications is that developers commonly obfuscate their code for security purposes. Obfuscation alters the bytecode of an application and presents difficulties when detecting code clones. The intuition behind ByClone is that detecting clones in obfuscated bytecode can be regarded as a similar bytecode detection problem. By calculating the similarity between two bytecode methods, we can determine whether they are clones in obfuscated or non-obfuscated formats.

Thus, in this paper, we propose a deep learning-based approach, ByClone, for source code similarity detection based on Java bytecode. Given a pair of methods in binary code, ByClone classifies the pair's corresponding source code as either clone or non-clone. We conduct our study using Android applications because they are widely available and many are obfuscated. Furthermore, the vulnerability and license incompatibility associated with the obfuscated bytecode format within the Java-based Android ecosystem are major concerns Lim et al. (2018).

1.2 Overview of the Methodology

We collect 140 Android apps from F-Droid and create a dataset of the similarity score for every pair of methods in bytecode. To build the dataset, we first run the state-of-the-art source code similarity detection tool, NiCad (Cordy & Roy, 2011), on every pair of source code methods. We chose NiCad because of its high precision and recall (95% and 96%, respectively). Then, we disassemble the Java `class` files to get the bytecode using the `javap` command line tool. We map the source code level similarity score that we got from NiCad to the corresponding method pairs in bytecode format. In other words, in our dataset, the similarity score of a pair of bytecode methods is their corresponding source code similarity score obtained from running NiCad.

ByClone leverages an LSTM Siamese Neural Network (Chicco, 2021a; Hochreiter & Schmidhuber, 1997) to learn and detect bytecode similarity. An LSTM Siamese Neural Network trains two LSTM sub-networks in parallel and concatenates them at the end to learn the similarity of the two inputs. ByClone is able to detect source code similarity in both regular and obfuscated bytecode. Among the 140 studied apps, we split them into training, validation, and test sets. The training set consists of 80% of the total number of applications (112 apps), and the testing and validation sets each consist of 10% of the total number of applications (14 apps each). We randomly split the apps based on the distribution of the clone method pairs to ensure all three sets have a similar ratio of clones/non-clones. We define a method pair as clones if they have a NiCad similar larger than 0.7 (Cordy & Roy, 2011).

1.3 Research Questions

We evaluate ByClone by answering the following research questions:

1.3.1 RQ1: Can we detect source code clones based on Java bytecode in Android applications?

We propose a deep learning-based approach to detect Java bytecode clones. Since embeddings are critical to the performance of a deep learning model, we compared the results of four popular embeddings: Word2Vec (Mikolov, Sutskever, Chen, Corrado, & Dean, 2013), FastText (Joulin, Grave, Bojanowski, & Mikolov, 2016), GloVe (Pennington, Socher, & Manning, 2014), and Instruction2Vec (Lee et al., 2019). We find that ByClone is able to detect clones in Java bytecode (i.e., similar score larger than 0.7 (Cordy & Roy, 2011)) with a precision and recall of 78.37 and 75.24, respectively, when using FastText as the embedding. ByClone outperforms the baselines that use traditional machine learning models such as SVM and random forest (precision is less than 1). Although most embeddings result in similar precision/recall, using Instruction2Vec is not able to detect any clones, which further highlights the importance of embeddings. Nevertheless, our findings show that ByClone is able to detect code clones based on the bytecode with relatively high precision and recall.

1.3.2 RQ2: Can we detect source code clones based on obfuscated bytecode?

Developers often obfuscate the bytecode, resulting in significant alteration to prevent others from decompiling the bytecode for privacy/security reasons (Su et al., 2017). We find that ByClone is able to accurately detect source code clones in obfuscated bytecode with a precision and recall of 82.55 and 70.95, respectively, when using Word2Vec as the embeddings. The results show the potential of using ByClone to detect clones based on obfuscated bytecode.

1.3.3 RQ3: Do configurations in R8 affect clone detection results?

R8 is the default code shrinker and obfuscation tool for Android applications. Developers often set custom configurations in R8 for their obfuscation and optimization requirements. The changes

in obfuscation and optimization defined by the R8 configuration can affect the resulting bytecode. We find that the presence of R8 configurations has an impact on ByClone's detection results. Particularly, the applications that contain pre-defined R8 configurations score a precision of 52.69 and a recall of 76.08. We remove the R8 configurations for the application and find that it results in worse performance results with a precision of 60 and a recall of 47.11. The performance comparison between the two cases indicates that our bytecode similarity detection approach is effective for obfuscated bytecode with R8 configurations.

1.4 Contributions

We summarize the contributions of this thesis as follows:

- Contribution 1. We train and evaluate 140 popular Java Android applications scraped from F-Droid. Our bytecode similarity dataset consists of pairs of bytecode methods and their source code similarity score provided by NiCad. This includes a total of 1.9 millions pairs of methods. We release this dataset and make it available online.

- Contribution 2. We present a deep learning approach to detect both unobfuscated and obfuscated bytecode in Java Android applications. We release the replication package for our Java bytecode similarity detection tool, ByClone.

- Contribution 3. We study the effects of R8 configurations on bytecode similarity detection. In particular, we study whether compiling Java bytecode with R8 options has an effect on the performance of our bytecode similarity detection tool, ByClone.

1.5 Thesis Organization.

Chapter 2 discusses the background of this thesis. Chapter 3 presents our approach. Chapter 4 discusses the data collection and model training. Chapter 5 presents the evaluation results. Chapter 6 presents a discussion of our thesis. Chapter 7 discusses the threats to validity. Chapter 8 summarizes the related work. Chapter 9 concludes the thesis.

Chapter 2

Background

Java is one of the most popular programming languages used by developers (Cass, 2015). The "write once, run anywhere" feature makes Java programs capable of being executed on a wide range of machines. The Java compiler first compiles the source code into bytecode (i.e., ".class" files). Then, the Java Virtual Machine (JVM) translates the bytecode to machine code according to the specific machine and executes the program; thus, making Java programs platform-independent. Java bytecode uses the instruction set that consists of an opcode to specify the operation to perform, and zero or more operands as the value to be operated on (Lindholm, Yellin, Bracha, & Buckley, 2013). Figure 2.1 shows an example of bytecode, where opcodes include `aload`, `invokevirtual`, and `return`, and operands consist of registers and constants.

Due to the characteristics of the Java compiler and Java bytecode, developers can use a decompiler to reverse-compile the source code from bytecode. A Java decompiler takes ".class" files as input and generates the corresponding source code. The decompiled Java source code is generally readable and similar to the original source code. However, such readable decompiled code poses risks to security, intellectual property, and copyright issues. The issues are more prevalent in Android, since all the applications, including both commercial and open source, are available from the Google Play store. Anyone can download the apk (Android Package Kit) files, which contain the bytecode of the application, and decompile the code.

Example Bytecode Before Obfuscation

```
protected void onCreate(android.os.Bundle);
  Code:
     0: aload_0
     1: aload_1
     2: invokespecial  #2    //comment
     5: new            #3    //comment
     8: dup
     9: invokespecial  #4    //comment
    12: ldc            #5    //comment
    14: invokevirtual  #6    //comment
    17: aload_0
    18: invokevirtual  #7    //comment
    21: ldc            #8    //comment
    23: invokevirtual  #9    //comment
    26: invokevirtual  #6    //comment
    29: invokevirtual  #10   //comment
    32: astore_2
    33: aload_0
    34: getfield       #11   //comment
    37: aload_2
    38: invokevirtual  #12   //comment
    41: return
```

Example Code After Obfuscation

```
protected void onCreate(android.os.Bundle);
  Code:
     0: aload_0
     1: aload_1
     2: invokespecial  #13   //comment
     5: new            #15   //comment
     8: dup
     9: invokespecial  #16   //comment
    12: astore_1
    13: aload_1
    14: ldc            #18   //comment
    16: invokevirtual  #22   //comment
    19: pop
    20: aload_1
    21: aload_0
    22: invokevirtual  #28   //comment
    25: ldc            #30   //comment
    27: invokevirtual  #36   //comment
    30: invokevirtual  #22   //comment
    33: pop
    34: aload_1
    35: invokevirtual  #40   //comment
    38: astore_1
    39: aload_0
    40: getfield       #44   //comment
    43: aload_1
    44: invokevirtual  #50   //comment
    47: return
```

Example Code Before Obfuscation (Decompiled)

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   String file_path = "file://" + this.getIntent().getStringExtra("file");
   this.wv.loadUrl(file_path);
 }
```

Example Code After Obfuscation (Decompiled)

```
protected void onCreate(Bundle var1) {
   super.onCreate(var1);
   StringBuilder var2 = new StringBuilder();
   var2.append("file://");
   var2.append(this.getIntent().getStringExtra("file"));
   String var3 = var2.toString();
   super.B.loadUrl(var3);
 }
```

Figure 2.1: An example of decompiled code before and after obfuscation.

build.gradle

```
android {
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    ...
}
```

Figure 2.2: An example of an build.gradle file.

To increase the difficulty of reading the decompiled code, developers use obfuscation before releasing Android apps to the Google Play store. Obfuscation is the process of transforming Java bytecode into a more complex, but semantically equivalent representation (Batchelder, 2007). Many obfuscation techniques exist, such as adding dummy files and methods, renaming variables and methods, and restructuring or shrinking the code while maintaining the same execution behaviour (Low, 1998). As a result, after decompiling the bytecode, the decompiled source code becomes very difficult to understand and almost unreadable. Figure 2.1 shows an example of a decompiled code snippet before and after the process of obfuscation. We can see that many variable names have changed, some new variables have been added, and the structure of the control flow has changed. Note that there are multiple configuration options for code obfuscation. In this example, for better readability, we keep the method names the same during obfuscation in the R8 configuration file.

R8 is a compiler that is able to shrink, optimize, and obfuscate the bytecode to improve efficiency and security (Google, 2023). R8 is popular among developers as it is part of the official Android software development kit (SDK) and is the default compiler for Android Studio 3.4 or higher. Android developers are able to customize the R8 configurations to the specific needs of the obfuscation and optimizations of an application. Some examples of configurations that a user can set in R8 include `keep` rules that notify the compiler which packages, classes or methods to not obfuscate or optimize, rules that allow R8 to change access modifiers, and rules that print the usage, seeds, and used configurations to a file, amongst many other rules. Developers are able to easily enable R8 and set their custom configurations in their Android applications by editing the build.gradle

proguard-rules.pro

```
-keepnames class * implements android.os.Parcelable {
        public static final android.os.Parcelable$Creator *;
}
-keepattributes InnerClasses,Signature,SourceFile,LineNumberTable
```

Figure 2.3: An example of an R8 configurations file.

file. Figure 2.2 shows the contents of a build.gradle file that has R8 enabled (`minifyEnabled` is set to true) and the R8 configurations file is either set to `proguard-android-optimize.txt` if the user chooses to optimize or `proguard-rules.pro` if the user chooses not to optimize. To define its configurations, R8 uses the rules files in Proguard, which was the default compiler Android Studio in versions prior to 3.4. (*ProGuard Manual: Usage | Guardsquare*, n.d.) Figure 2.3 shows an example of a R8 configuration file. In this Proguard rules file, the first rule denotes that every class that implements the Parcelable interface should be prevented from being obfuscated or optimized. The second rule indicates that all inner classes, signatures, source files, and line numbers are also prevented from obfuscation or optimization. Adding R8 configurations affects the compilation outcome of the bytecode, which could have an impact on the efficacy of bytecode clone detection.

Detecting source code clones based on bytecode has great potential for practitioners to identify and avoid issues such as unconsciously introducing vulnerable code or third-party libraries with improperly licensed code. However, to the best of our knowledge, there is little research in this direction due to a lack of clone datasets. Hence, in this paper, we collected and released a code clone benchmark for Java bytecode. We also proposed an approach, ByClone, as one of the first steps toward detecting source code clones based on both obfuscated and unobfuscated Java bytecode.

Chapter 3

Approach

In this chapter, we discuss our approach, ByClone, to detect clones in bytecode. We formulate clone detection as a bytecode similarity detection problem. Given two pieces of bytecode (whether obfuscated or not), our approach uses the Siamese Neural Network to determine whether their corresponding source code methods are clones (i.e., a binary classification).

Figure 3.1 shows an overview of ByClone and the experiments. We first pre-process the bytecode to remove the constants and dynamic values. The pre-processed bytecode is then fed into a pre-trained embedding layer to create input vectors for the deep learning model. Then, we train a Siamese Neural Network that models the similarities between two pieces of bytecode. The outputs of the two parallel Siamese Neural Network layers are concatenated. Once concatenated, the output is trained on multiple drop-out, batch normalization, and dense layers. The final layer is a

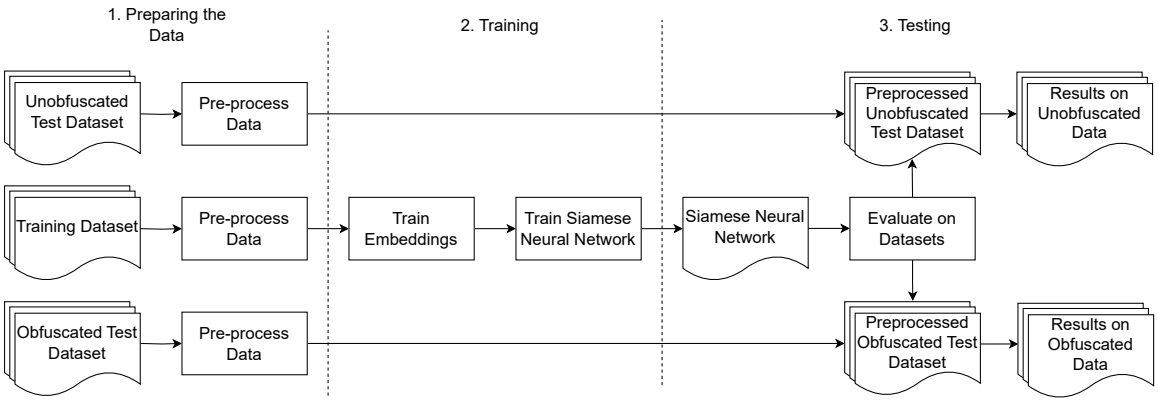

Figure 3.1: An overview of our approach.

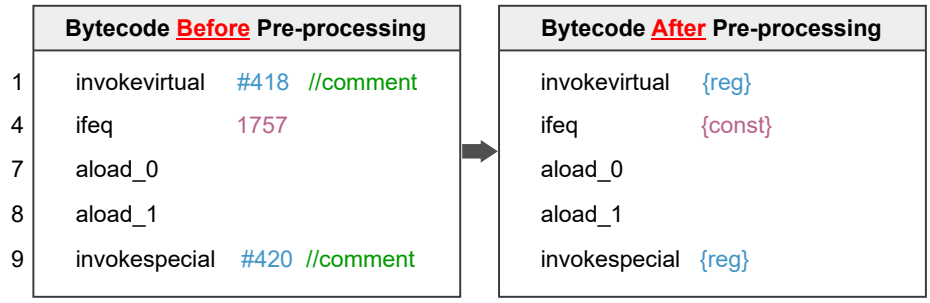

Figure 3.2: An example of our Java bytecode preprocessing step.

one-neuron dense layer that outputs a score from 0 to 1. We set the threshold to 0.7 by following NiCad's recommendation (Cordy & Roy, 2011; NiCad, 2022). Note that we conduct our study at the method level granularity, so the inputs to ByClone are the bytecode of a pair of methods. Below, we discuss each step in detail.

3.1 Pre-processing Bytecode

Bytecode may contain some constants or dynamic values that can affect code similarity detection. For example, as shown in Figure 3.2, the bytecode produced by the disassembler contains some automatically generated comments that document specific metadata. These comments are automatically generated by the Java disassembler and serve to document specific metadata. The # symbol followed by a number denotes a constant pool entry in Java bytecode. The constant pool is a table-like data structure that holds constant values. These constants include numeric literals, strings, class names, field names, and method signatures. Lastly, the bytecode contains some constants whose values are directly transferred from the source code. Since our goal is to detect bytecode similarity, we remove such values and comments by preprocessing the bytecode. Specifically, we apply the following preprocessing steps by following a prior study (Zuo et al., 2018):

(1) Replace constant pool indices (denoted by the symbol # followed by a numeric value) with a special token `{reg}`.

(2) Replace constant values with a special token `{const}`.

(3) Remove comments inside the bytecode.

(4) Replace switch keys and switch blocks from the "tableswitch" and "lookupswitch" instructions with `{switch_key}` and `{switch_block}`, respectively.

(5) Replace exception table row elements that represent the "from", "to", and "type" values to `{from}`, `{to}`, and `{type}`, respectively.

As shown in Figure 3.2, after preprocessing, the constants and constant pool indices are replaced with the placeholder tokens, and the comments are removed. We apply preprocessing to both the inputs of our Siamese Neural Network and to the bytecode corpus used as input for the embedding layer, as described below.

3.2 Embedding Layer

To prepare for the inputs to the Siamese Neural Network, we convert the preprocessed bytecode into a vector. Each dimension in the vector represents a unique token in the preprocessed bytecode, and each element represents the frequency of a token in a given bytecode. The vectors are then fed into an embedding layer, which performs an automated feature selection that converts the input vectors into a lower-dimension vector (Mikolov, Chen, Corrado, & Dean, 2013; Rodriguez & Spirling, 2022). Embedding is crucial to ByClone because it allows effective comparison of code snippets for similarity. Prior studies (Ding, Li, Shang, & Chen, 2022; Kang, Bissyandé, & Lo, 2019) found that different embeddings may have an effect on the downstream tasks. Therefore, in this paper, we also explore the effect of different embeddings on bytecode similarity detection. In particular, we consider four embeddings that are commonly used in the literature (Z. Chen & Monperrus, 2019; Ding et al., 2022; Perone, Silveira, & Paula, 2018):

Word2Vec. Word2Vec (Mikolov, Sutskever, et al., 2013) is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. Given the bytecode, Word2Vec maps each token to a vector representation, which is then leveraged as the context (i.e., neighboring tokens) to predict a target token. The underlying intuition is to cluster the vectors of similar tokens together in a vector space. Through training, Word2Vec can then predict the target token based on past appearances. Word2Vec considers the bytecode in our data as plain text.

GloVe. GloVe (Pennington et al., 2014) is an unsupervised learning algorithm for obtaining vector representations for words. Using only non-zero elements, GloVe leverages a word to word co-occurrence matrix. It is able to achieve high performance by utilizing the count data and capturing the linear substructures of a word vector space. GloVe also considers the input bytecode in out model as plain text.

FastText. FastText (Joulin et al., 2016) is a lightweight library for learning of word embeddings and text classification created by Facebook's AI Research lab. FastText utilizes the skipgram model, in which every word is represented as a bag of character n-grams and every n-gram is represented by a vector. Like Word2Vec and GloVe, FastText also interprets bytecode as plain text.

Instruction2Vec. Instruction2Vec (Lee et al., 2019) is a framework to vectorize instructions of the assembly language. Instruction2Vec uses Word2Vec to generate a lookup table for all the library functions, opcodes, registers, and hex values. Then, it maps them onto a fixed-dimension table that reflects the syntax of the assembly code. While Instruction2Vec inputs assembly code and not bytecode, both languages share syntactical similarities.

3.3 Siamese Neural Network

Our bytecode similarity detection technique uses the Siamese Neural Network to train our model. A Siamese Neural Network (SNN) is a neural network architecture that contains two or more identical sub-networks for computing the similarity between inputs (Chicco, 2021b). Figure 3.3 shows a diagram of our SNN model. The input layer concurrently takes in two tokenized and preprocessed methods in bytecode, which are fed into the embedding layer to learn or convert to the embeddings. The output of the embeddings is then trained with a batch normalization layer, which standardizes the inputs and improve the stability and speed of the training process (Ioffe & Szegedy, 2015; T. Kim, 2021; Santurkar, Tsipras, Ilyas, & Madry, 2018). The output of the batch normalization layer is used to train the Bi-directional Long short-term memory (Bi-LSTM) layer. We set the number of units in each Bi-LSTM layer to 300 (300 dimensions of hidden state) and use `tanh` as the activation function since it is found to be more effective in LSTM (Farzad, Mashayekhi,

& Hassanpour, 2019). Both outputs of the Bi-LSTM layer are trained with a soft attention mechanism, which helps improve the accuracy of the predictions (Q. Chen et al., 2017; Yao et al., 2015). The output of the attention layer from one of the sub-networks is subtracted from, multiplied with and concatenated with the output of the Bi-LSTM layer of the other sub-network and vice versa. The two concatenated layers are then trained with an additional Bi-LSTM layer to improve model accuracy. Then, a one-dimensional global average pooling layer and a one-dimensional global max pooling layer are applied to the two subnetworks, and the results are concatenated. The average and max pooling layers help downsample our data and reduce spatial dimensions while retaining essential information. (Bieder, Sandkühler, & Cattin, 2021) The outputs of the two parallel networks are then merged through the concatenation layer. To stabilize the concatenated vectors, the output is trained with a batch normalization layer. Additionally, the output from the batch normalization layer is trained with two iterations of the following sequence of layers: a dense layer with 300 neurons and an Exponential Linear Units (ELU) activation function, a batch normalization layer, and a dropout layer. The ELU activation function allows for both fast learning and good generalization performance (Clevert, Unterthiner, & Hochreiter, 2015). We set the dropout rate to 0.5 to reduce the risk of overfitting (T. Kim, 2021; Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014; T. Zhang, Gao, Ma, Lyu, & Kim, 2019). Finally, the output from the dropout layer will be the input into a dense layer. The dense layer uses a sigmoid activation function. It will converge the neurons from the last layer and output a single number in the range of 0 to 1, which will be converted to be either *non-clone* or *clone* depending on whether the value is below or above the set threshold (we set the threshold value to 0.98). Our model is inspired by the winning team's solution to the "Quora Question Pairs" contest on Kaggle (DataCanary et al., 2017). This contest aims to find the best prediction model to identify duplicate questions on Quora, a question-and-answer forum. (*Quora*, n.d.) The winning team utilizes an enhanced LSTM (Q. Chen et al., 2017) to achieve the best results in the competition.

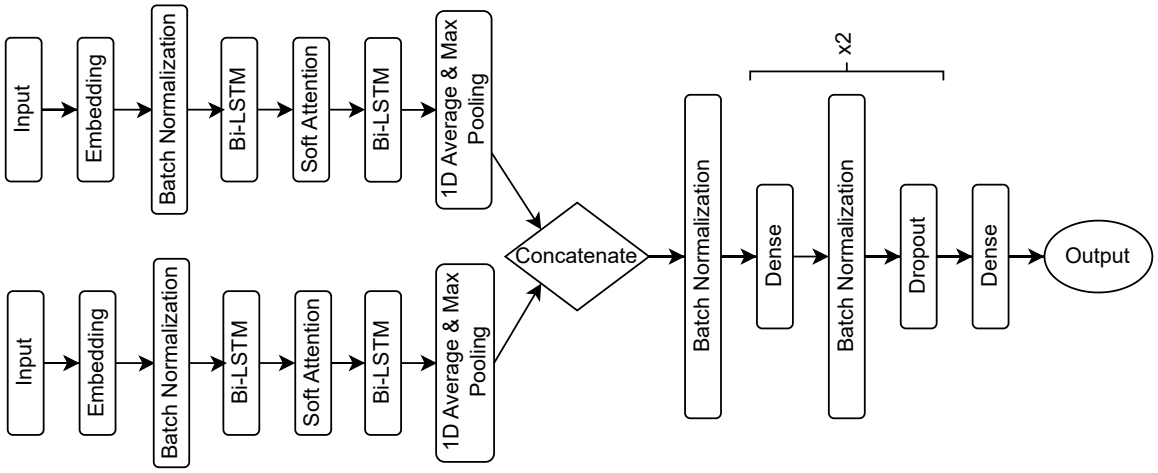

Figure 3.3: ByClone Model Diagram

3.4 Implementation Details

We performed our experiments using CUDA 12.3 on a server with a NVIDIA Tesla A100 Ampere GPU. To train our model, we used the Adam optimizer (Kingma & Ba, 2014), which is an effective and commonly used optimization algorithm in deep learning with a learning rate of 1e-3. Our model trains on 50 epochs and we set the batch size to 64 since our training data is large in size. We compile our model with a binary cross-entropy loss function. Since our output is expected to be a binary score of either 0 or 1, we chose to utilize a loss function that is effective for training binary classification models. We describe in detail on how we obtain the ground truth to train the model in Section 4. Running our model takes around 5 minutes each for the un-obfuscated dataset and the obfuscated dataset, which contain the same 14 applications.

Chapter 4

Experimental Setup

In this chapter, we describe the experimental setup of our study, including the data collection process for training the embedding layers and the Siamese Neural Network.

4.1 Building the Ground Truth Dataset

Our goal is to detect, given two pieces of bytecode (whether or not they are obfuscated), what the similarity of their corresponding source code is and whether they are clones or not. Therefore, we need to have a mapping between the bytecode and the corresponding source code, and the similarity score between source code to train the Siamese Neural Network. We use a source code clone detection tool called NiCad (Roy & Cordy, 2008) to build our ground truth. NiCad computes the similarity score between two code snippets by comparing the hybrid language-sensitive text of the code snippets. We choose NiCad because of its high precision (95%) and recall (96%) in detecting near-miss intentional clones (Roy & Cordy, 2008, 2009), but other clone detection tools can also be used. We use its latest available version upon the submission of this paper (i.e., Nicad 6.2 (NiCad, 2022)) to build the dataset.

NiCad allows users to set a threshold to only report code pairs that have a certain similarity score. The default threshold setting for NiCad is 0.3, which means that code pairs with a similarity score of 0.7 or above are detected as clones. However, for our purposes, we wanted to examine the similarity of all possible code pairs to train our model. Therefore, we maximized the threshold to

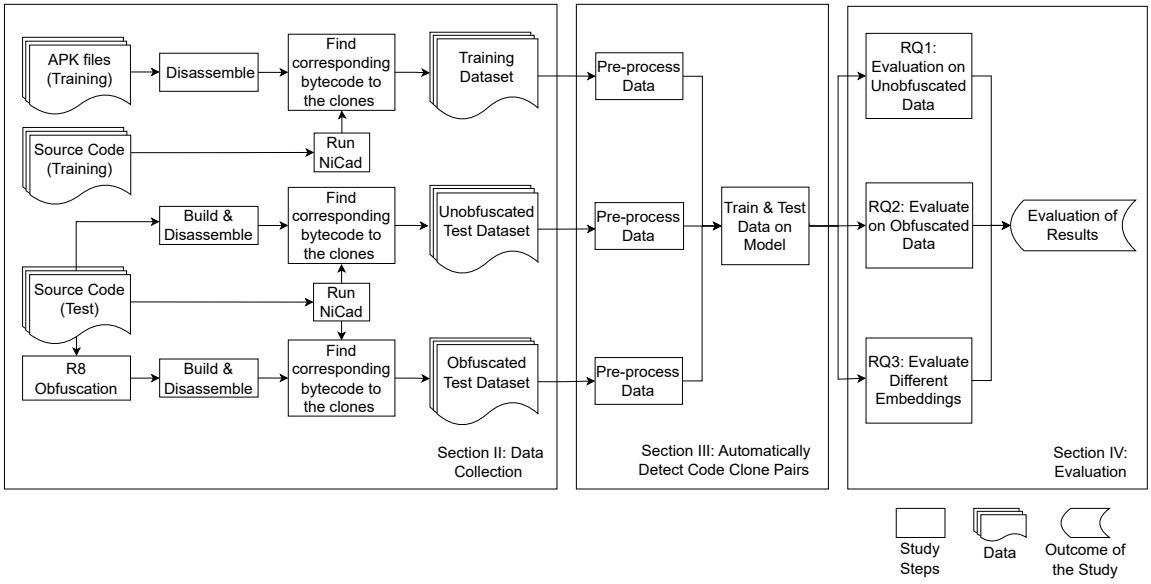

Figure 4.1: The overall process of this paper.

Table 4.1: An Overview of the Studied Android Applications.

	# Stars	LOC	# Commits	# Clone Pairs	# Non-Clone Pairs
Max	25.6k	175.5k	454.4K	629	316.7K
Average	1.1k	23.2K	11.9k	33.8	12.7K
Median	317.5	15.5K	1.8k	5	1.5k
Total	152.8k	3.2M	1.7M	4.7k	1.8M

1.0 so that code pairs with 0 to 100% similarity were detected. We set the granularity of detected clones to the method level. After running NiCad, we get the similarity scores between all pairs of methods associated with an application. Similar to NiCad and other clone detection tools (Roy & Cordy, 2008; Saha, Roy, Schneider, & Perry, 2013), the final similarity score can be used to determine whether or not two pieces of bytecode are clones by setting a threshold.

4.2 Collecting the Studied Android Apps

Due to the lack of datasets on bytecode similarity, we created a Java Android application dataset that contains the similarity score between obfuscated and unobfuscated code pairs. Our dataset consists of Android applications from F-Droid, which is a repository of Free and Open Source Software (FOSS) for the Android platform. (*F-Droid Free and Open Source Android App Repository*, n.d.). We are able to obtain the information of repositories from index.xml of F-Droid. Figure A.1 shows

an example application in the F-Droid index.xml file. We obtain both the source code and the APK file of the Android applications for our dataset. The source code is needed for building the ground truth and the APK file is needed to extract the bytecode. F-Droid provides the link of the source code repository from which the source code is downloaded. The applications on F-Droid are hosted on a variety of source code repositories, including Github, Gitlab, and Bitbucket. We studied the apps that are hosted on GitHub. We selected the apps with the most number of stars, discarded the apps that do not have any APK files, and the ones that are actively maintained (had commits in 2022 or later). Our query retrieves the Android applications and their GitHub repository link. There were a total of 4,194 applications available of F-Droid. After filtering out the applications based on the above-mentioned criteria, we are left with 140 applications. Table 4.1 shows the statistics of the studied applications. On average, these applications have 1.1k stars, 23.2K in LOC, and over 11.9k commits. We use these 140 applications for training and evaluating ByClone.

4.3 Generating the Bytecode and Obtaining Their Similarities

Once we have collected the source code of the Android applications, we compile the applications and retrieve the bytecode for both *normal* and *obfuscated* builds. We discuss our build process in detail below.

Extracting Bytecode from the Android Applications. We split the 140 collection of Android applications into 112 for training, 14 for validation, and 14 for testing, based on the commonly used 80-10-10 ratio to split datasets (Jain & Meenu, 2021; D. Kim & MacKinnon, 2018). The bytecode from both our training and validation datasets are obtained by disassembling the APK files of the 126 applications. APK files contain already-built bytecode, including ones that might be obfuscated. Hence, our training and validation datasets contain a mix of bytecode that is unobfuscated or obfuscated. There are a total of 63 applications with `minifyEnabled` set to true (to shrink, obfuscate, and optimize the bytecode), and the remaining 63 applications either have it set to false or do not specify that option in the `build.gradle` file.

Since some applications contain more clones than others, we split the applications by considering the number of clones in each application such that the training, validation, and test set also

contain roughly 80%, 10%, and 10% of the number of clones, respectively. There are a total of 63 applications with `minifyEnabled` set to true (to shrink, obfuscate, and optimize the bytecode), and the remaining 63 applications either have it set to false or do not specify that option in the `build.gradle` file.

Since our test dataset also consists of a mix of obfuscated and unobfuscated code, we wish to ensure that the same applications are used for the evaluation of both the unobfuscated and obfuscated data to effectively compare the performance of the two sets. On the other hand, we build the applications in our test dataset from the source code, instead of using the APK file to obtain our bytecode. This allows us to study and compare the performances of both unobfuscated and obfuscated bytecode as well as obfuscated bytecode with R8 (an Android code shrinker) configurations. We build the Android applications in the test dataset using Gradle, the default build system for Android, with two settings: unobfuscated (i.e., regular builds) and obfuscated. For the applications with R8 configurations, we later removed the R8 configurations and built again. After the regular build process, we obtain both the source code (`.java`) and the unobfuscated bytecode (`.class`) files. For the obfuscated build, we use R8 to compile the Android applications during the build process and generate obfuscated bytecode. For both the unobfuscated and the obfuscated build, we disassemble the resulting .class files to Java bytecode using `javap`, a Java class file disassembler command line tool. Specifically, we use the `javap` tool with the `-c` flag to print out disassembled code comprised of Java bytecode instructions.

Mapping Source Code Similarity Detection Results to Bytecode. As discussed above, we use NiCad's source code similarity detection results as the ground truth. The results contain the similarity score (ranges from 0 to 100) for all pairs of methods. We rely on method names to map from the bytecode to the source code, and use the source code similarity scores as the ground truth for the similarity among bytecode. Specifically, we use the method signatures to find the matching methods between source code and bytecode. We save the results in an XML file that stores bytecode clone pairs. For every bytecode pair in the XML file, we store the following information: 1) the bytecode of the first method, 2) the bytecode of the second method, and 3) the similarity score between the first and second method.

4.4 Training Java Bytecode Embeddings

Since there is no existing pre-trained embedding on Java bytecode to the best of our knowledge, we construct a corpus of bytecode instructions from 10 Android projects and the top 10 most popular non-Android Java projects on GitHub. To avoid biases, these 10 Android projects do not overlap with the projects that we use for training/evaluating ByClone. We choose both Android and regular Java projects to increase the diversity in the bytecode so that it is not limited to only Android projects. After preprocessing, we gathered a total of 1,967,146 instructions in our corpus. We then train the different embedding models from the corpus. We use 300 as the size of the embedding vector by following a prior study (Gu, Tandon, Ahn, & Radicchi, 2021). These embedding models generate a vectorized dictionary of the words in the corpus. The vectors and values from the embedding layer are then used as inputs to the Siamese neural network. We release the information for the applications that we use for training and evaluating the embeddings in our replication package, which can be accessed online (*ByClone*, n.d.).

4.5 Resampling the Data

There are significantly more non-clone code pairs than clone pairs in our dataset, which creates a skewed dataset that makes it difficult for the model to accurately learn the features of the under-represented class. In total, there are 4.7k code pairs that have a similarity score larger than 0.7 and 1.8M code pairs that have a similarity score smaller than 0.7. In order to address this imbalance, we implement random resampling to balance our training dataset. We do so by reducing the quantity of non-clone pairs (code pairs whose similarity score is smaller than 0.7). We resample the data to a 1:10 clone-to-non-clone ratio by reducing the non-clone sample quantity to 10 times the number of clone sample quantity. Undersampling the majority class allows our dataset to be more balanced, which allows the model to be trained more accurately (Pereira, Costa, & Silla Jr., 2021). Additionally, training our model would require less memory and resources as a result. After the resampling process, we have 3,236 clones and 32,360 non-clones (1:10) code pairs in our training dataset.

Chapter 5

Evaluation

In this chapter, we first present the metrics that we use to evaluate ByClone, which classifies if a given pair of bytecode methods are clones. Then, we present the motivation, approach, and results of the research questions (RQs). Given two bytecode snippets, ByClone predicts whether or not two code snippets are clones. Hence, we use the following commonly used metrics to evaluate how well ByClone is able to detect clones: Precision, Recall, and F-score.

Precision. Precision measures the ratio of correctly classified positive values to the total number of predicted positive values (i.e., a code pair predicted to be clone is actually clone). Precision is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

where a higher precision value indicates more accurate predictions. The advantage of precision is that it is a more suitable metric for assessing the performance of imbalanced datasets than accuracy (Juba & Le, 2019).

Recall. Recall measures the ratio of correctly classified positive values to the total number of positive values. Recall is defined as:

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

where a higher recall value indicates the model can correctly classify more clone code pairs. Recall is also a more appropriate metric for evaluating the performance of imbalanced datasets than accuracy (Juba & Le, 2019).

F-1 Score. The F-1 score represents the harmonic mean of the precision and recall (Tharwat, 2020). F-1 Score is defined as:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \tag{3}$$

where a higher F-1 score indicates a more accurate classification.

5.1 RQ1: Can we detect source code clones based on Java bytecode in Android applications?

Motivation. Software reuse is a common practice due to the wide availability of open-source and commercial libraries. Many developers use code from third-party software and only have access to the bytecode and not the source code. This can cause developers to unknowingly use bytecode that contains licensing agreements. These licensing agreements can further be unwittingly breached by the developer and lead to legal or financial ramifications. As a first step to tackle the above-mentioned challenges, finding clones in bytecode is crucial. Therefore, in this RQ, we wish to evaluate the performance of ByClone in detecting clones based on bytecode.

Approach. As mentioned in Chapter 4.3, we train ByClone by splitting the 140 studied Android applications into 112 applications for training and 14 each for validation and testing. This is based on the 80-10-10 ratio that is commonly used in splitting datasets into training, validation, and test sets (Jain & Meenu, 2021; D. Kim & MacKinnon, 2018). We consider the distribution of clones when splitting the apps to reduce classification biases. We then apply the data resampling technique described in Chapter 4.5 to increase the ratio between clone and non-clone method pairs in the training set. We first evaluate the overall clone detection accuracy using the above-mentioned evaluation metrics. Prior studies (Z. Chen & Monperrus, 2019; Ding et al., 2022; Perone et al., 2018) found that the embeddings in deep learning models affect the results of downstream tasks. Hence, in this

Table 5.1: Overall Results of ByClone.

Obfuscation	Model	Precision	Recall	F Score
Un-obfuscated	Decision Tree	0.51	14.10	0.99
	Gradient Boosting Classifier	0.45	0.38	0.41
	Logistic Regression	0.12	0.76	0.20
	Random Forest	0.93	0.19	0.32
	Stochastic Gradient Descent Classifier	0.46	18.29	0.90
	SVM	0.00	0.00	0.00
	ByClone-FastText	**78.37**	75.24	**76.77**
	ByClone-GloVe	69.17	71.81	70.47
	ByClone-Instruction2Vec	0.00	0.00	0.00
	ByClone-Word2Vec	76.86	**76.57**	76.72
Obfuscated	Decision Tree	0.90	21.43	1.74
	Gradient Boosting Classifier	0.00	0.00	0.00
	Logistic Regression	0.08	0.48	0.14
	Random Forest	0.96	0.24	0.38
	Stochastic Gradient Descent Classifier	0.40	13.81	0.77
	SVM	0.00	0.00	0.00
	ByClone-FastText	71.71	70.00	70.84
	ByClone-GloVe	71.21	**75.95**	73.50
	ByClone-Instruction2Vec	0.00	0.00	0.00
	ByClone-Word2Vec	**82.55**	70.95	**76.31**

RQ, we also evaluate the performance of different embeddings in ByClone. We compare the results from ByClone with six baseline models: decision tree, gradient boosting classifier, logistic regression, random forest, stochastic gradient descent classifier, and support vector machine (SVM). We train and evaluate these baseline models using the same dataset that we use for ByClone.

Result. ***We find that ByClone is able to achieve significantly higher precision and recall (78.37 and 75.24 respectively) compared to the baseline models where the highest precision and recall are 0.93 and 18.29.*** Table 5.1 shows the clone detection results. ByClone with the highest F-1 Score can achieve a precision of 78.37 and a recall of 75.24 when detecting clones. The result is significantly higher than the baseline models, for which the highest F-1 Score is only 0.99. We also find that the embeddings have a large impact on the clone detection results, although the embeddings are trained on the same instruction corpus. In particular, the model trained using Instruction2Vec shows the worst result, where the model could not detect any clones. The ByClone model that is trained using Instruction2Vec is not able to converge and predicts all the pairs to be non-clones. The

Table 5.2: Clone detection results for different clone types.

Obfuscation	Clone Type	Precision	Recall	F Score
Un-obfuscated	Non-Clones	99.93	99.93	99.93
	Type 2 & 3 Clones	68.32	69.23	68.77
	Type 1 Clones	78.52	79.05	78.79
Obfuscated	Non-Clones	99.90	99.95	99.93
	Type 2 & 3 Clones	74.60	60.66	66.91
	Type 1 Clones	89.71	83.47	86.49

reason may be that while Instruction2Vec aims to capture the syntax of assembly code, it might not be able to capture the syntax of bytecode effectively (Lee et al., 2019). Among the four evaluated embeddings, FastText gave the best results (precision and recall are 78.37 and 75.24) when detecting clones in unobfuscated bytecode. While FastText produces the highest F-1 Score (76.77) among the embeddings, it is worth noting that the F-1 Score of Word2Vec (76.72) is only slightly worse than that of FastText. GloVe also performs comparably to both Word2Vec and FastText with an F-1 Score of 70.47. All three embeddings (FastText, Word2Vec, and GloVe) are widely used embeddings and are effective for various natural language processing (NLP) tasks. (Torregrossa, Allesiardo, Claveau, Kooli, & Gravier, 2021) The efficacy of these three approaches in training word embeddings in NLP tasks extends to training bytecode instructions for our tool, ByClone.

We find that code pairs with a similarity score close to the cutoff threshold have lower precision and recall. Figure 5.1 shows the distribution of the prediction results across NiCad's code similarity scores categorized into true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The results are obtained from one of the best performing embeddings, Word2Vec. The distribution of both of the incorrectly classified sets (FP and FN) is close to NiCad's 70% similarity score threshold. This finding suggests the code clones that are around the cutoff threshold may have a greater probability of being misclassified. Hence, we further evaluate how well ByClone can detect different types of clones: Type-1 (exactly the same) and Type-2&3 (also called near-miss clones), where the clones that have some differences such as variable names or some statements added/deleted (Roy, 2009) (in Table 5.2). In Type-2 clones, the code is syntactically the same with differences in variable names, string values, and styling. However, since when the source code is

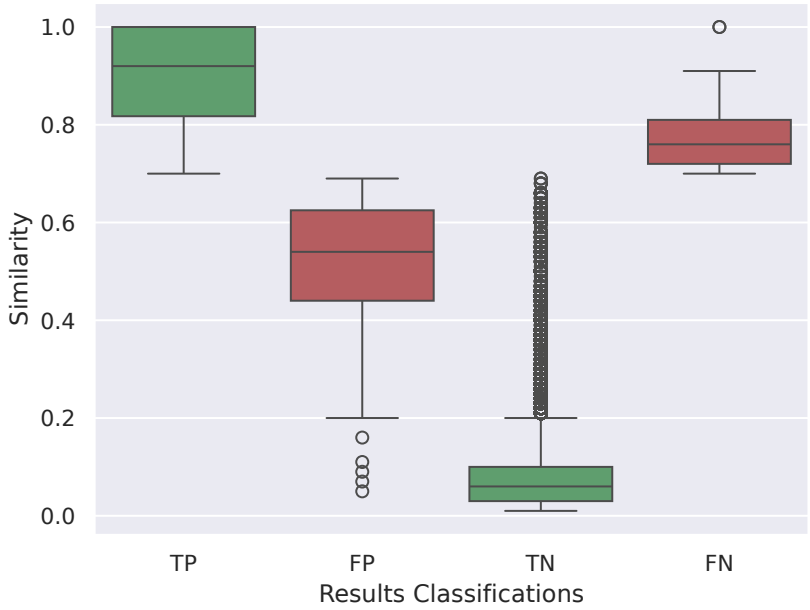

Figure 5.1: Distribution of the classification results across NiCad's code similarity scores.

compiled to bytecode, information such as variable names is lost, we consider Type-2 and Type-3 clones together. We can see that Type-1 clones have a higher precision and recall compared to Type-2 and Type-3, although we still achieve relatively good results in all clone types.

We find that ByClone is able to achieve considerably higher precision and recall (78.37 and 75.24, respectively) compared to the baseline models (best precision and recall are less than 1 and 20, respectively). ByClone is more effective in detecting Type-I clones.

5.2 RQ2: Can we detect source code clones based on obfuscated bytecode?

Motivation. Java application developers often obfuscate their code in order to protect the source code and prevent others from reverse engineering the application. Obfuscation transforms the source code without impacting its functionality. For example, R8, the default bytecode obfuscation tool in Android Studio, supports identifier and package renaming to make the bytecode more difficult to decompile or understand (Zhan et al., 2020, 2021). We built ByClone to detect clones in Android

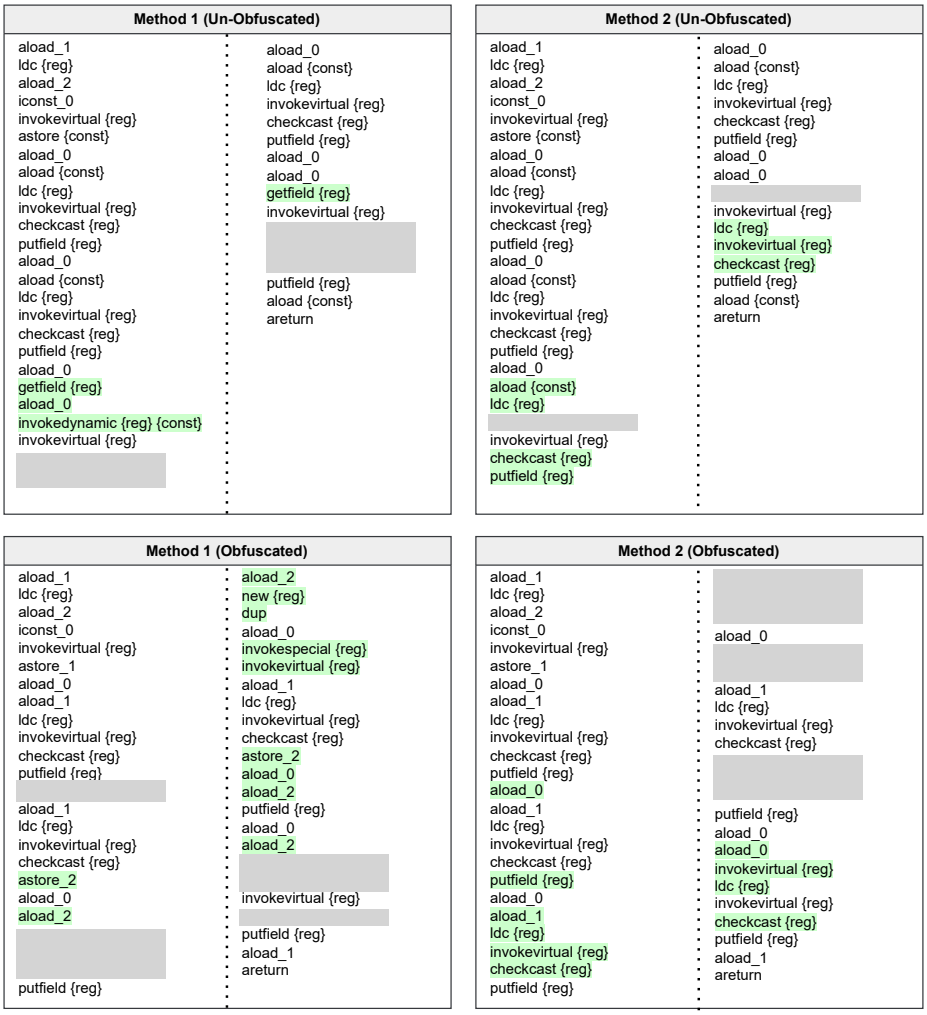

Figure 5.2: A comparison of a code pair misclassified in obfuscated code, but correctly classified in unobfuscated code.

applications that can be obfuscated. In this RQ, we evaluate the performance of ByClone on obfuscated bytecode.

Approach. We utilize the same training (112 applications) and validation (14 applications) datasets we use for RQ1. For our test dataset, we utilize the same 14 applications as we use for the RQ1 test dataset. However, we obfuscate these same applications to evaluate the effect of ByClone on obfuscated Java bytecode. We have detailed how we choose these applications in Chapter 4. We obfuscated our Java Android applications with Android Studio's built-in obfuscation tool named R8. Once we built the files, we obtain the APK files for the Android projects. We then extract the `.class` files from the APK and further disassemble the `.class` files to Java bytecode files. We do not add more obfuscated bytecode in the training data because, as mentioned in Section 4, it already contains both obfuscated and unobfuscated bytecode. We run NiCad on the source code of both our training and test datasets to find similarity between methods. We then match the code clone pairs to their bytecode equivalent code. Our model then makes a prediction on whether two bytecode methods are a clone. Depending on the predicted and actual results of the similarity scores, we are able to calculate the precision, recall, and F-1 scores

Result. ***ByClone is able to achieve a higher precision but lower recall (82.55 and 70.95, respectively) when detecting clones in obfuscated bytecode.*** We observe that the best-performing embedding, Word2Vec, demonstrates a higher precision (82.55) and lower recall (70.95) in the obfuscated dataset compared to RQ1. This suggests that ByClone-Word2Vec has a higher false negative rate, but a lower false positive rate for detecting clones in obfuscated bytecode. The false negative rate can be attributed to the altered content of the pairs of bytecode methods as a result of obfuscation. In other words, two bytecode segments that share syntactical similarities in their unobfuscated form can be become dissimilar once obfuscated. Figure 5.2 shows an example of a method pair that is incorrectly classified in our obfuscated dataset as a non-clone (false negative), but is correctly classified as a clone (true positive) in the unobfuscated dataset. The source code of these two methods was classified as a clone with 81% similarity by NiCad. Similar to the source code, the unobfuscated bytecode of the two methods share many similarities. On the other hand, the obfuscated bytecode shares fewer similarities, as shown in the figure, since many details were removed/shrunk. However, these misclassified results only constitute a small fraction of all the classification results.

In particular, there are only 31 false negative results in the obfuscated dataset that are classified as true positive results in the unobfuscated dataset out of the 525 total shared clones in the test datasets.

While the three embeddings have similar results, Word2Vec achieves the highest F-1 score. We also find that there is variation in the scores across the different embeddings. As shown in Table 5.1, the scores of the different embeddings for ByClone range from 0 to 82.55 in precision and 0 to 75.95 in recall. The embeddings in the obfuscated dataset follow a similar trend in RQ1 where Word2Vec, GloVe, and FastText all perform comparatively well, with the best-performing embedding, Word2Vec, having a precision and recall of 82.55 and 70.95, respectively. Instruction2Vec performs the worst with a precision and recall of 0. Instruction2Vec classifies all the results as non-clones and, therefore, achieves very low precision and recall scores. On the other hand, both GloVe and FastText perform relatively well with GloVe scoring a precision and recall of 71.21 and 75.95 respectively, and with FastText scoring a precision of 71.71 and a recall of 70.00. The finding suggests that the Word2Vec embedding can achieve higher performance in detecting clones for obfuscated bytecode. Moreover, similar to RQ1, traditional machine learning algorithms (e.g., decision tree and SVM) are not effective in detection clones in obfuscated bytecode, with the highest F-1 Score of only 1.74.

Observing the misclassified results, we find that a few source code clones could be overlooked by a bytecode similarity detection tool due to the dissimilarity of the bytecode. We conduct randomly sample the prediction results across different classification types. In particular, we randomly select five samples each from the true positive, true negative, false positive, and false negative results. Figure B.1 shows one of the five randomly selected true positive results that is calculated to have a 80% similarity by NiCad. We observe that both the bytecode and source codes of the two methods are similar and ByClone is able to correctly identify the pair as a clone. Figure B.2 is one of the randomly selected true negative results that is calculated to have a 7% similarity by NiCad. We notice that both the bytecode and source codes of the two methods are very dissimilar and ByClone is able to correctly classify the two methods as non-clones. On the other hand, we analyze the results of the randomly sampled false positive and false negative values to understand the reason ByClone incorrectly classifies some results. Figure B.3 shows a false positive result obtained from

Table 5.3: A comparison of results from applications with R8 configurations and without.

Pre-existing R8 Configurations	Obfuscated	Compiled With Configurations	Precision	Recall	F Score
Yes	Yes	Yes	70.89	45.16	55.17
		No	60.00	47.11	52.78
	No	-	69.07	71.18	70.11
No	Yes	No	85.82	81.76	83.74
	No	-	83.28	80.74	81.99

the sampling that is assigned a similarity score of 61% and is classified as a non-clone by NiCad. However, not only the bytecode, but also the source code of the methods is shown to be similar. All five randomly sampled false positive results follow the same pattern, where both the source code and bytecode were similar, but was assigned a similarity score close to, but below the 70% similarity threshold. This finding is in agreement with findings from figure 5.1, which show that the FP results are skewed towards the 70% cutoff threshold. Figure B.4 shows a sample of a false negative result with NiCad a similarity score of 77%. We notice that the source code of the two methods are quite similar, but the bytecode of the pair share very little similarity. This suggests that a few source code clones could be overlooked by a bytecode similarity detection tool due to the dissimilarity of the bytecode.

ByClone is able to achieve significantly higher precision and lower recall (82.55 and 70.95, respectively) on obfuscated bytecode compared to the unobfuscated ones. We find that obfuscation may remove details in the bytecode that can affect clone detection results.

5.3 RQ3: Do configurations in R8 affect bytecode clone detection results?

Motivation. Android application developers are able to customize the obfuscation configurations in R8 based on their obfuscation requirements. R8 configurations can be customized with proguard rules and allow developers to preserve certain code segments (classes, members, etc). Since customizing R8 configurations changes the obfuscation and optimization processes, the output bytecode can be altered and might affect the efficacy of bytecode similarity detection approach. Thus, in this RQ, we wish to investigate the sensitivity of ByClone on R8's configurations.

Approach. We use the same training and validation dataset set as in RQ1 and RQ2. Additionally, we use the same 14 applications in both RQ1 and RQ2 and split them according to whether they have R8 configurations already written in the code. We do so by reviewing the proguard rules file in all 14 applications and checking whether there are any pre-written rules. From the 14 total applications in our test dataset, 5 contain proguard rules and 9 do not. For our evaluation, we first run ByClone on the 5 applications that contain proguard rules. Then, we run ByClone on the 9 applications that do not contain any proguard rules. This allows us to compare the efficacy of ByClone on applications both with and without R8 configurations from the dataset we collected. We additionally evaluate ByClone on the 5 applications that do contain proguard rules *after removing the pre-existing proguard rules and obfuscating them without the rule*. We remove the proguard rules to compare the effects of R8 configuration on bytecode similarity within the same applications. We evaluate ByClone on the three aforementioned subsets of the test dataset (i.e., apps containing R8 configurations, apps that do not contain R8 configurations, and apps with R8 configurations that were removed). All three results are obfuscated to evaluate the effect of R8 configurations on similarity detection. We additionally evaluate the unobfuscated results on both the subset of applications that have R8 configurations and those that do not.

Results. ***Removing R8 configurations slightly decreases the performance in applications that have pre-existing R8 options.*** Table 5.3 shows the results of the bytecode similarity detection on: 1) the applications in our test dataset that already contained R8 configurations in their code, 2) the applications in our test dataset that did not contain any R8 configurations, 3) the same applications from (1), but with the R8 options removed and obfuscated, 4) the unobfuscated results of the applications with R8 configurations, and 5) the unobfuscated results from the applications without R8 configurations. We observe that the set of applications that do not have any pre-existing R8 configurations perform significantly better than the ones with configurations. The applications without configurations achieved a precision score of 85.82 and a recall of 81.76. On the other hand, the applications with configurations had precision and recall scores of 70.89 and 45.16, respectively. The considerable difference between the performance of the two subsets leads us to speculate whether the presence of R8 configurations significantly diminishes the bytecode similarity detection performance. Since these two subsets consist of different sets of applications, we decided to additionally

evaluate the scores of the unobfuscated bytecode methods of the two subsets. We found that the difference in performance metrics was still present. Namely, the subset with pre-existing R8 configurations (precision 69.07 and recall 71.18) also performed worse than the subset without pre-existing R8 configurations (precision 83.28 and recall 80.74) in their unobfuscated formats. Since these two subsets contain different applications and their performance scores vary, we compare the results of R8 configurations with the same set of applications to ensure a more accurate comparison of the prediction results. This entails removing the existing R8 configurations from the proguard rules file from the applications that contain configurations. However, removing the R8 configuration resulted in worse performance than keeping the configurations. The results of the applications with R8 configurations removed are 60 for precision and 47.11 for recall. *Nevertheless, we find that ByClone's result is not very sensitive to R8 configurations.*

The applications containing R8 options perform slightly better than when the options are removed from the same applications. Upon further investigation, we observe that the sets with options and options removed have different numbers of bytecode method pairs. The set with options has 55,084 pairs of methods, and the set with options removed has 12,525. This is likely because many R8 configurations prevent specified code from being removed. Hence, the set with R8 configurations will have more pairs of methods. In order to more accurately compare the performance of the two sets, we selected all the pairs that the two sets have in common and calculated the performance metrics for this subset of data. This results in a precision of 69.05% and recall of 43.94% for the set with R8 options, and 64.44% precision and 43.93% recall for the set with R8 options removed. The set with the R8 configurations performs slightly better than the set with options removed, and produces fewer false positive results. The reason might be that many R8 configurations used by developers are `keep` options that prevent bytecode from obfuscation, which might result in more effective predictions from ByClone since there are less diverse in the bytecode patterns.

ByClone is not very sensitive to R8 configurations, where the results are similar before/after removing the configurations.

Chapter 6

Discussion

For Researchers. Our results showed that ByClone was able to detect clones in both unobfuscated and obfuscated Java bytecode effectively compared to the six baseline models. We also found that a significant impact in the performance of ByClone was the choice in embeddings, and the different R8 configurations have an effect to the performance. We encourage future research to design more effective approaches to take into account the different aspects of Java bytecode clone detection. Though our model was able to detect clones in obfuscated Java bytecode effectively, there were some pairs of code that were misclassified. Especially for those cases that have a close to 70% NiCad similarity threshold, clones are commonly misclassified. Such cases can be further studied to improve the accuracy of Java bytecode clone detection.

For Practitioners. One application of Java bytecode clone detection is to facilitate downstream tasks for practitioners. When risky Java bytecode is identified, ByClone can be used to help scan among a collection of artifacts to identify similar code. Practitioners and tool developers can evaluate our proposed approach in their target use cases, for example, detecting vulnerabilities in Java Android applications, especially when the bytecode is obfuscated. Another example is to identify potential issues in software licensing compliance in a proactive manner. We also provide insights into the effectiveness of Java bytecode clone detection in both unobfuscated and obfuscated bytecode, and in various R8 configurations. These build options are commonly used by Android application developers. Future tooling efforts can integrate clone detection functionality into the build

process, thus preventing issues (e.g., vulnerability and licensing issues) in an early stage.

Chapter 7

Threats to Validity

External Validity. Our study is conducted on open-source Java Android applications from F-Droid. Our findings might not generalize to commercial systems or other applications. Future studies may consider validating our findings by conducting an extensive study on additional systems. Our model is trained on 126 Android applications and evaluated on 14 Android applications. In future work, this number can be further extended to test on more applications to produce even more robust results. Namely, the analysis on the effect of R8 configurations could be evaluated on more applications. Although some Java source code clones appear similar at the source level, they may differ once compiled into bytecode. This discrepancy could affect the performance of our model. Nevertheless, our model predicts source code clones based on bytecode well as evident in our precision and recall scores.

Internal Validity. We use a different bytecode extraction process for the training/validation datasets and the test dataset. For the training and validation sets, we use bytecode disassembled from the APK files. This ensures our training reflects real-world Android application data and helps us obtain data from a large number of Android application efficiently. However, the test dataset is built from the source code because the test dataset needs to be evaluated on both unobfuscated and obfuscated code. We further discuss this threat in chapter 7. However, we obfuscate with R8, an obfuscation tool built-in to the Android SDK. In fact, half (63 applications) of our training and validations sets have R8 enabled in their `build.gradle` files. Therefore, we believe the training

and validation datasets are still representative of our test dataset. Moreover, ByClone performs well when trained with our training and validation sets and evaluated with our test set in both unobfuscated and obfuscated bytecode, which further suggests the different bytecode extraction processes for the datasets do not negatively affect the performance of our model. Instruction2Vec yields a precision and recall of 0 for both the un-obfuscated and obfuscated clones as it classifies all the results as non-clones. This issue might be due to the over-representation of non-clones in our dataset and Instruction2Vec's ability to capture the semantics of assembly code, which differs from bytecode.

Construct Validity. Our model is trained on 126 Android applications and evaluated on 14 Android applications. In future work, this number can be further extended to test on more applications to produce even more robust results. Namely, the analysis on the effect of R8 configurations could be evaluated on more applications. In our current test dataset, we discovered 5 applications with R8 options, which could make it more difficult reach a clear conclusion on the effect of R8 configuration on bytecode similarity detection. However, even with 5 Android applications, we are able to produce a high number of method pairs. We produce 55,084 method pairs from the these 5 applications compiled with R8 configurations and 12,525 method pairs from the same applications compiled without R8 configurations.

Chapter 8

Related Work

Detecting Code Clone Similarity in Source Code. Many prior works have focused on detecting similarity in source code. As a result, many clone detection tools are proposed at the source code level. They leverage the structure and syntax of the source code to identify code clones. For example, NiCad is a scalable and flexible clone detection tool that we use in our study to build the ground truth clone dataset at the source code level. It uses flexible pretty printing and code normalization to convert code into a standard textual format (Roy & Cordy, 2008). Another popular clone detection tool is CCFinder, which uses a multi-linguistic token-based approach (Kamiya, Kusumoto, & Inoue, 2002). In addition, there are deep learning-based clone detection tools, such as CCLearner, that train a deep learning classifier from known method-level code clones and nonclones (L. Li, Feng, Zhuang, Meng, & Ryder, 2017). White et al. (White, Tufano, Vendome, & Poshyvanyk, 2016) proposed a deep learning clone detection approach that leverages code patterns at both the lexical and syntactic levels. For a more systematic review of clone detection tools, readers are suggested to refer to Ain et al. (Ain, Butt, Anwar, Azam, & Maqbool, 2019).

Detecting Code Clone Similarity in Binary Code. Detecting binary code similarity is a relatively new field where the binary code is analyzed to identify similar patterns. Approaches are proposed to detect binary code similarity (Haq & Caballero, 2021). For example, Zhu et al. proposed a

graph search algorithm to efficiently search for small-sized similar binary code in a large-sized program (Zhu, Jiang, Chen, & Yan, 2021). BinDeep is a deep learning-based approach that combines both CNN and LSTM to measure the similarity of two binary functions (Tian et al., 2021). They use the instruction sequences in the binary code and vectorize the instructions as embeddings. Li et al. proposed a multi-semantic feature fusion attention network to detect binary code similarity, with the benefits of both learning the overall features and capturing the relationships between features (B. Li et al., 2023).

Detecting Code Clone Similarity in Java Bytecode. While limited, there exist some studies that detect clones in Java bytecode using various techniques. For example, Keivanloo et al. proposed an approach that uses two criteria to detect similarities in bytecode: pattern similarity using Semantic Web querying and reasoning and content similarity using the Jaccard coefficient (Keivanloo, Roy, & Rilling, 2012). Additionally, Keivanloo et al. have expanded on their initial approach to build a scalable bytecode clone detection and search model that applies semantic-enabled token matching. (Keivanloo, Roy, & Rilling, 2014). Another study uses the Smith-Waterman algorithm, which is an algorithm derived from gene sequence matching, to align bytecode sequences (Yu, Yang, Chen, & Chen, 2019).

While prior research in code similarity detection has focused on source code or C/C++ assembly code for their analysis (Liu, 2021; Yang et al., 2021), our approach introduces an approach to detect similarity in Java bytecode. Although there exists some, yet limited, prior research on bytecode similarity detection, our approach differs from existing methods in its methodology, dataset used, and ability to detect clones in obfuscated bytecode. In particular, we employ a deep learning approach that detects source code clones based on Java bytecode. Additionally, we create our own dataset from Android applications available on F-Droid and we detect not only regular bytecode code clones, but also obfuscated bytecode code clones.

Code obfuscation is a common practice for Android application development (Dong et al., 2018; Maiorca, Ariu, Corona, Aresu, & Giacinto, 2015; Wang & Rountev, 2017; X. Zhang, Breitinger, Luechinger, & O'Shaughnessy, 2021). The Android Studio desktop application has an obfuscation tool named R8 built into the Android software development kit (SDK). Many developers use obfuscation for various reasons, including security and plagiarism prevention. Our work proposes a

bytecode similarity detection model that not only detects code clone similarity in the unobfuscated bytecode from Java Android applications, but also the obfuscated bytecode from Java Android applications.

Chapter 9

Conclusion

The use of third-party Java software, whether in its original or obfuscated bytecode form, may pose a greater risk of vulnerability to attacks and potential issues with software licensing compliance, due to the lack of access to the source code. In this paper, we propose a clone detection approach, ByClone, to detect source code clones based on bytecode for both unobfuscated and obfuscated bytecode. Due to a lack of existing data, we collected and conducted our study on 140 Java Android applications. We obtained the pairs of methods and their similarity scores by utilizing NiCad, a source code similarity detection tool, and mapped the scores to the corresponding bytecode to create the ground truth. ByClone uses a Siamese Neural Network, where it takes in a pair of methods in bytecode format and classifies whether they are clones or not. We built our Java Android applications both with a normal configuration and an obfuscation configuration. Results showed that the model performs well on both unobfuscated and obfuscated datasets. We also compare the performance of different popular embeddings on our model. Results showed that the Word2Vec embedding outperformed all the other embedding models in the obfuscated dataset, whereas the FastText embedding produced the best performance results in the unobfuscated dataset. Additionally, we explore the effects of R8 (Android bytecode obfuscator) configuration settings. We find that ByClone is not sensitive on R8 configurations.

Appendix A

An Example of FDroid index.xml

index.xml

```
<application id="net.gsantner.markor">
    <id>net.gsantner.markor</id>
    <added>2017-09-11</added>
    <lastupdated>2023-10-15</lastupdated>
    <name>Markor</name>
    <summary>Text editor - Notes & ToDo. Lightweight. Markdown and atodo.txt support.</summary>
    <desc>📝 Create notes and manage your to-do list using simple markup formats 🌲 ....
        <a href="fdroid.app:org.documentfoundation.libreoffice">LibreOffice</a>) or to-do apps ....
        <a href="fdroid.app:com.owncloud.android">OwnCloud</a>,
        <a href="fdroid.app:com.nextcloud.client">NextCloud</a>,
        <a href="fdroid.app:com.seafile.seadroid2">Seafile</a>,
        <a href="fdroid.app:com.nutomic.syncthingandroid">Syncthing</a>,
        <a href="fdroid.app:org.amoradi.syncopoli">Syncopoli</a> 👀 These apps may also ....
        <b>Support the project:</b>
        <a href="https://github.com/gsantner/markor/issues/new/choose">Report ideas and issues</a> |
        <a href="https://github.com/gsantner/markor/discussions">Join community discussion</a> |
        <a href="https://crowdin.com/project/markor/invite">Translate</a> |
        <a href="https://github.com/gsantner/markor#contributions">More information about contributions</a>
    </desc>
    <license>Apache-2.0</license>
    <categories>Writing</categories>
    <category>Writing</category>
    <web/>
    <source>https://github.com/gsantner/markor</source>
    <tracker>https://github.com/gsantner/markor/issues</tracker>
    <changelog>https://github.com/gsantner/markor/blob/HEAD/CHANGELOG.md</changelog>
    <author>Gregor Santner</author>
    <marketversion>2.11.1</marketversion>
    <marketvercode>148</marketvercode>
    <package>
        <version>2.11.1</version>
        <versioncode>148</versioncode>
        <apkname>net.gsantner.markor_148.apk</apkname>
        <srcname>net.gsantner.markor_148_src.tar.gz</srcname>
        <hash type="sha256">e393a87975b7fe3aee74b878f66e0db0f3091ad859f6d5ae3419ff3e573aa60b</hash>
        <size>11113846</size>
        <sdkver>16</sdkver>
        <targetSdkVersion>33</targetSdkVersion>
        <added>2023-10-15</added>
        <sig>af12ddae122db68f2ebf5469dbb98620</sig>
        <permissions>INSTALL_SHORTCUT,INTERNET,MANAGE_EXTERNAL_STORAGE,...</permissions>
    </package>
    <package>
        <version>2.10.9</version>
        ...
    </package>
    <package>
        <version>2.10.8</version>
        ...
    </package>
</application>
```

Figure A.1: An Example Application in F-Droid.

Appendix B

Example Results of ByClone on Obfuscated Bytecode

Source Code of First Method

```
public byte getByte() {
        try {
                int numByteRead = commonRead(1);
                if (numByteRead == 1) {
                        return buffer[0];
                } else {
                        Log.e(TAG, "Error in reading byte");
                }
        } catch (IOException e) {
                e.printStackTrace();
        }
        return -1;
}
```

Source Code of Second Method

```
public long getLong() {
        try {
                int numByteRead = commonRead(4);
                if (numByteRead == 4) {
                        return ByteBuffer.wrap(Arrays.copyOfRange(buffer, 0, 4))
                                                .order(ByteOrder.LITTLE_ENDIAN).getInt();
                } else {
                        Log.e(TAG, "Error in reading byte");
                }
        } catch (IOException e) {
        e.printStackTrace();
        }
        return -1;
}
```

Bytecode of First Method

```
aload_0
iconst_1
invokespecial {reg}
iconst_1
if_icmpne {const}
aload_0
getfield {reg}
iconst_0
baload
ireturn
```



```
ldc {reg}
ldc {reg}
invokestatic {reg}
pop
goto {const}
astore_1
aload_1
invokevirtual {reg}
iconst_m1
ireturn
Exception table:
{from} {to} {target} java/io/IOException
{from} {to} {target} java/io/IOException
```

Bytecode of Second Method

```
aload_0
iconst_4
invokespecial {reg}
iconst_4
if_icmpne {const}
aload_0
getfield {reg}
iconst_0
iconst_4
invokestatic {reg}
invokestatic {reg}
getstatic {reg}
invokevirtual {reg}
invokevirtual {reg}
i2l
lreturn
ldc {reg}
ldc {reg}
invokestatic {reg}
pop
goto {const}
astore_1
aload_1
invokevirtual {reg}
ldc2_w {reg}
lreturn
Exception table:
{from} {to} {target} java/io/IOException
{from} {to} {target} java/io/IOException
```

Figure B.1: An example of a true positive result.

Figure B.2: An example of a true negative result.

Source Code of First Method

```
public void startCTMU(int cRange, int trim, int tgen) {
    if (tgen == -1) tgen = 1;
    try {
        mPacketHandler.sendByte(mCommandsProto.COMMON);
        mPacketHandler.sendByte(mCommandsProto.START_CTMU);
        mPacketHandler.sendByte(cRange | (tgen << 7));
        mPacketHandler.sendByte(trim);
        mPacketHandler.getAcknowledgement();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Source Code of Second Method

```
public void setSqrs(int wavelength, int phase, int highTime1, int highTime2,
                    int prescalar) {
    if (prescalar == -1) prescalar = 1;
    try {
        mPacketHandler.sendByte(mCommandsProto.WAVEGEN);
        mPacketHandler.sendByte(mCommandsProto.SET_SQRS);
        mPacketHandler.sendInt(wavelength);
        mPacketHandler.sendInt(phase);
        mPacketHandler.sendInt(highTime1);
        mPacketHandler.sendInt(highTime2);
        mPacketHandler.sendByte(prescalar);
        mPacketHandler.getAcknowledgement();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Bytecode of First Method

```
iload_3
istore {const}
iload_3
iconst_m1
if_icmpne {const}
iconst_1
istore {const}
aload_0
getfield {reg}
aload_0
getfield {reg}
getfield {reg}
invokevirtual {reg}
aload_0
getfield {reg}
aload_0
getfield {reg}
getfield {reg}
invokevirtual {reg}
aload_0
getfield {reg}
iload {const}
bipush {const}
ishl
iload_1
ior
invokevirtual {reg}
aload_0
getfield {reg}
iload_2
invokevirtual {reg}
aload_0
getfield {reg}
[redacted]
invokevirtual {reg}
pop
goto {const}
astore {const}
aload {const}
invokevirtual {reg}
return
Exception table:
{from} {to} {target} java/io/IOException
```

Bytecode of Second Method

```
iload {const}
istore {const}
iload {const}
iconst_m1
if_icmpne {const}
iconst_1
istore {const}
aload_0
getfield {reg}
aload_0
getfield {reg}
getfield {reg}
invokevirtual {reg}
aload_0
getfield {reg}
aload_0
getfield {reg}
getfield {reg}
invokevirtual {reg}
aload_0
getfield {reg}
[redacted]
iload_1
[redacted]
invokevirtual {reg}
aload_0
getfield {reg}
iload_2
invokevirtual {reg}
aload_0
getfield {reg}
iload_3
invokevirtual {reg}
aload_0
getfield {reg}
iload {const}
invokevirtual {reg}
aload_0
getfield {reg}
iload {const}
invokevirtual {reg}
aload_0
getfield {reg}
invokevirtual {reg}
pop
goto {const}
astore {const}
aload {const}
invokevirtual {reg}
return
Exception table:
{from} {to} {target} java/io/IOException
```

Figure B.3: An example of a false positive result.

Figure B.4: An example of a false negative result.

References

Ahtiainen, A., Surakka, S., & Rahikainen, M. (2006). Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises. In *Proceedings of the 6th baltic sea conference on computing education research: Koli calling 2006* (pp. 141–142).

Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., & Maqbool, B. (2019). A systematic review on code clone detection. *IEEE access*, *7*, 86121–86144.

Batchelder, M. R. (2007). *Java bytecode obfuscation* (Unpublished doctoral dissertation). McGill University, Montréal, QC.

Bieder, F., Sandkühler, R., & Cattin, P. C. (2021). Comparison of methods generalizing max-and average-pooling. *arXiv preprint arXiv:2103.01746*.

Byclone. (n.d.). https://anonymous.4open.science/r/ByClone-D141/. (Accessed: 2024-02-24)

Cass, S. (2015). The 2015 top ten programming languages. *IEEE Spectrum, July*, *20*.

Chen, Q., Zhu, X., Ling, Z.-H., Wei, S., Jiang, H., & Inkpen, D. (2017). Enhanced lstm for natural language inference. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: Long papers)*. Association for Computational Linguistics. Retrieved from http://dx.doi.org/10.18653/v1/P17-1152 doi: 10.18653/v1/p17-1152

Chen, Z., & Monperrus, M. (2019). *A literature study of embeddings on source code.*

Chicco, D. (2021a). Siamese neural networks: An overview. In H. Cartwright (Ed.), *Artificial neural networks* (pp. 73–94). New York, NY: Springer US. Retrieved from https://doi.org/10.1007/978-1-0716-0826-5_3 doi: 10.1007/978-1-0716-0826-5_3

Chicco, D. (2021b). Siamese neural networks: An overview. *Artificial neural networks*, 73–94.

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Cordy, J. R., & Roy, C. K. (2011). The nicad clone detector. In *2011 ieee 19th international conference on program comprehension* (p. 219-220). doi: 10.1109/ICPC.2011.26

DataCanary, hilfialkaff, Jiang, L., Risdal, M., Dandekar, Nikhil, & tomtung. (2017). *Quora question pairs.* Kaggle. Retrieved from https://kaggle.com/competitions/quora-question-pairs

Ding, Z., Li, H., Shang, W., & Chen, T.-H. P. (2022, mar). Can pre-trained code embeddings improve model performance? revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering*, *27*(3). Retrieved from https://doi.org/10.1007/s10664-022-10118-5 doi: 10.1007/s10664-022-10118-5

Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., ... Zhang, K. (2018). Understanding android obfuscation techniques: A large-scale investigation in the wild. In *Security and privacy in communication networks: 14th international conference, securecomm 2018, singapore, singapore, august 8-10, 2018, proceedings, part i* (pp. 172–192).

Farzad, A., Mashayekhi, H., & Hassanpour, H. (2019, jul). A comparative performance analysis of different activation functions in lstm networks for classification. *Neural Computing and Applications*, *31*(7), 2507–2521. Retrieved from https://doi.org/10.1007/s00521-017-3210-6 doi: 10.1007/s00521-017-3210-6

F-droid free and open source android app repository. (n.d.). https://f-droid.org/. (Accessed: 2023-10-30)

Google. (2023). *D8 dexer and r8 shrinker.* https://r8.googlesource.com/r8. (Accessed: 2023-09-15)

Gu, W., Tandon, A., Ahn, Y.-Y., & Radicchi, F. (2021, Jun 18). Principled approach to the selection of the embedding dimension of networks. *Nature Communications*, *12*(1), 3772. Retrieved from https://doi.org/10.1038/s41467-021-23795-5 doi: 10.1038/s41467-021-23795-5

Haq, I. U., & Caballero, J. (2021). A survey of binary code similarity. *ACM Computing Surveys*

(CSUR), *54*(3), 1–38.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735-1780. doi: 10.1162/neco.1997.9.8.1735

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd international conference on machine learning* (p. 448–456).

Jain, P., & Meenu. (2021). Recognition of mechanical tools using artificial neural network. In *Recent advances in mechanical engineering: Select proceedings of itme 2019* (pp. 637–644).

Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

Juba, B., & Le, H. S. (2019, Jul.). Precision-recall versus accuracy and the role of large data sets. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 4039-4048. Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/5193 doi: 10.1609/aaai.v33i01.33014039

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. doi: 10.1109/TSE.2002.1019480

Kang, H. J., Bissyandé, T. F., & Lo, D. (2019). Assessing the generalizability of code2vec token embeddings. In *2019 34th ieee/acm international conference on automated software engineering (ase)* (p. 1-12). doi: 10.1007/s00521-017-3210-6

Keivanloo, I., Roy, C. K., & Rilling, J. (2012). Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *2012 6th international workshop on software clones (iwsc)* (pp. 36–42).

Keivanloo, I., Roy, C. K., & Rilling, J. (2014). Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, *95*, 426-444. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167642313002773 (Special Issue on Software Clones (IWSC'12)) doi: https://doi.org/10.1016/j.scico.2013.10.006

Kim, D., & MacKinnon, T. (2018). Artificial intelligence in fracture detection: transfer learning from deep convolutional neural networks. *Clinical radiology*, *73*(5), 439–445.

Kim, T. (2021). *Generalizing mlps with dropouts, batch normalization, and skip connections.*

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lee, Y., Kwon, H., Choi, S.-H., Lim, S.-H., Baek, S. H., & Park, K.-W. (2019). Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, *9*(19). Retrieved from https://www.mdpi.com/2076-3417/9/19/4086 doi: 10.3390/app9194086

Li, B., Zhang, Y., Peng, H., Fan, Q., He, S., Zhang, Y., ... Ma, A. (2023). Multi-semantic feature fusion attention network for binary code similarity detection. *Scientific Reports*, *13*(1), 4096.

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)* (p. 249-260). doi: 10.1109/ICSME.2017.46

Lim, K., Han, J., Kim, B., Cho, S.-J., Park, M., & Han, S. (2018, 09). Open-source android app detection considering the effects of code obfuscation. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, *9*, 50-61. doi: 10.22667/JOWUA.2018.09.30.050

Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2013). *The java virtual machine specification*. Addison-wesley.

Liu, Z. (2021). Binary code similarity detection. In *2021 36th ieee/acm international conference on automated software engineering (ase)* (pp. 1056–1060).

Low, D. (1998, apr). Protecting java code via code obfuscation. *XRDS*, *4*(3), 21–23. Retrieved from https://doi.org/10.1145/332084.332092 doi: 10.1145/332084.332092

Maiorca, D., Ariu, D., Corona, I., Aresu, M., & Giacinto, G. (2015). Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, *51*, 16–31.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). *Distributed representations of words and phrases and their compositionality.*

Mlouki, O., Khomh, F., & Antoniol, G. (2016). On the detection of licenses violations in the android

ecosystem. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 1, p. 382-392). doi: 10.1109/SANER.2016.73

NiCad. (2022). *Nicad clone detector.* https://www.txl.ca/txl-nicaddownload.html. (Accessed: 2023-08-29)

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (emnlp)* (pp. 1532–1543).

Pereira, R. M., Costa, Y. M., & Silla Jr., C. N. (2021). Toward hierarchical classification of imbalanced data using random resampling algorithms. *Information Sciences*, *578*, 344-363. Retrieved from https://www.sciencedirect.com/science/article/pii/S0020025521007234 doi: https://doi.org/10.1016/j.ins.2021.07.033

Perone, C. S., Silveira, R., & Paula, T. S. (2018). *Evaluation of sentence embeddings in downstream and linguistic probing tasks.*

Proguard manual: Usage | guardsquare. (n.d.). https://www.guardsquare.com/manual/configuration/usage/. (Accessed: 2024-02-24)

Quora. (n.d.). https://www.quora.com/. (Accessed: 2024-02-20)

Ragkhitwetsagul, C., Krinke, J., & Clark, D. (2018). A comparison of code similarity analysers. *Empirical Software Engineering*, *23*, 2464–2519.

Rodriguez, P. L., & Spirling, A. (2022). Word embeddings: What works, what doesn't, and how to tell the difference for applied research. *The Journal of Politics*, *84*(1), 101–115.

Roy, C. K. (2009). Detection and analysis of near-miss software clones. In *2009 ieee international conference on software maintenance* (p. 447-450).

Roy, C. K., & Cordy, J. R. (2008). Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th ieee international conference on program comprehension* (p. 172-181).

Roy, C. K., & Cordy, J. R. (2009). A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Second international conference on software testing verification and validation, ICST 2009, denver, colorado, usa, april 1-4, 2009, workshops proceedings* (pp. 157–166). IEEE Computer Society. Retrieved from https://dblp.org/rec/

conf/icst/RoyC09.bib doi: 10.1109/ICSTW.2009.18

Saha, R. K., Roy, C. K., Schneider, K. A., & Perry, D. E. (2013). Understanding the evolution of type-3 clones: An exploratory study. In *2013 10th working conference on mining software repositories (msr)* (p. 139-148).

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 31). Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, *15*(1), 1929–1958.

Su, F.-H., Bell, J., Kaiser, G., & Baishakhi, R. (2017). *Deobfuscating android applications through deep learning.*

Tang, W., Luo, P., Fu, J., & Zhang, D. (2020). Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 ieee 27th international conference on software analysis, evolution and reengineering (saner)* (pp. 104–115).

Tharwat, A. (2020). Classification assessment methods. *Applied computing and informatics*, *17*(1), 168–192.

Tian, D., Jia, X., Ma, R., Liu, S., Liu, W., & Hu, C. (2021). Bindeep: A deep learning approach to binary code similarity detection. *Expert Systems with Applications*, *168*, 114348. Retrieved from https://www.sciencedirect.com/science/article/pii/S0957417420310332 doi: https://doi.org/10.1016/j.eswa.2020.114348

Torregrossa, F., Allesiardo, R., Claveau, V., Kooli, N., & Gravier, G. (2021). A survey on training and evaluation of word embeddings. *International Journal of Data Science and Analytics*, *11*(2), 85–103.

Wang, Y., & Rountev, A. (2017). Who changed you? obfuscator identification for android. In *2017 ieee/acm 4th international conference on mobile software engineering and systems (mobilesoft)* (pp. 154–164).

White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In *Proceedings of the 31st ieee/acm international conference on automated software engineering* (pp. 87–98).

Yang, J., Fu, C., Liu, X.-Y., Yin, H., & Zhou, P. (2021). Codee: a tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering*, *48*(7), 2224–2244.

Yao, L., Torabi, A., Cho, K., Ballas, N., Pal, C., Larochelle, H., & Courville, A. (2015). Video description generation incorporating spatio-temporal features and a soft-attention mechanism. *arXiv preprint arXiv:1502.08029*, *6*(2), 201–211.

Yu, D., Yang, J., Chen, X., & Chen, J. (2019). Detecting java code clones based on bytecode sequence alignment. *IEEE Access*, *7*, 22421-22433. doi: 10.1109/ACCESS.2019.2898411

Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., ... Liu, Y. (2020). Automated third-party library detection for android applications: Are we there yet? In *Proceedings of the 35th ieee/acm international conference on automated software engineering* (pp. 919–930).

Zhan, X., Liu, T., Liu, Y., Liu, Y., Li, L., Wang, H., & Luo, X. (2021). A systematic assessment on android third-party library detection tools. *IEEE Transactions on Software Engineering*, *48*(11), 4249–4273.

Zhang, T., Gao, C., Ma, L., Lyu, M., & Kim, M. (2019). An empirical study of common challenges in developing deep learning applications. In *2019 ieee 30th international symposium on software reliability engineering (issre)* (p. 104-115).

Zhang, X., Breitinger, F., Luechinger, E., & O'Shaughnessy, S. (2021). Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, *39*, 301285.

Zhu, X., Jiang, L., Chen, Z., & Yan, L. (2021). Searching for similar binary code based on the influence of basic block vertex. In *2021 3rd international conference on advances in computer technology, information science and communication (ctisc)* (pp. 17–24).

Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., & Zhang, Z. (2018). Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*.

Đurić, Z., & Gasevic, D. (2013). A source code similarity system for plagiarism detection. *The*

Computer Journal, *56*(1), 70–86.