Detection of Android Applications Homology Based on the Authors’ Programming Style

Jiayu Xu
Beijing haidian district west TuCheng Road 10, Beijing university of posts and telecommunications.
isxujiayu@foxmail.com

Abstract. As the mobile internet playing an increasingly important role in people’s daily life, the copyright issue of Android applications attaches more and more attentions. Because Android applications are vulnerable to reverse engineering, there are many criminals illegally modify and re-sign the official android for their own interests, which has seriously damaged the user’s rights and disturbed the order of the mobile internet. In this paper, a novel detects method of Android applications similarity based on the authors’ programming style is described. In the proposed method, we firstly extract a series of basic features that can represent the programming style of author. Considering that the features selected manually are highly subjective, we tried ICA, PCA and LDA to optimize the features. Ultimately, an Extreme Learning Machine (ELM) model is trained with sequences of feature vectors. The results of experiments demonstrate that our proposed method has better performance than string comparison and grammar tree analysis when fighting against variable substitution, insert independent statement and data stream confusion.

1. Introduction
Nowadays as smartphones and other mobile devices spread, more and more entertainment and work that originally required computers to be migrated to mobile devices. According to China Internet Development Report 2018, as of the end of 2017, the number of mobile internet users in China reached 753 million, and the proportion of mobile internet users reached 97.5%. Mobile phones have become the most important mobile internet devices [1]. Android applications provide great convenience for people’s lives, people can use mobile devices to shop, play games and more. However, unlike iOS applications, Android applications are easily decompiled, resulting from most Android applications are written in JAVA. As the “2015 Android mobile phone application piracy research report” shows, in the sample of 10305 genuine applications, every genuine android application corresponds to 92 pirated copies [2]. In this case, software homology detection is extremely important. There are some methods for software homology detection that have been proposed. These methods can be roughly classified into four categories: string-based software homology detection, token-based software homology detection, graph-based software homology detection, and syntax-tree-based software homology detection. However, the plagiarism technology is also constantly evolving. Edward L.Jones [3] raised the 10-difficulty level of plagiarism. At the level II, it just simply modifies the comment statements. Nevertheless, when it comes to the level X, it uses the equivalent control structure to replace the original structure, in which case it is very difficult to detect plagiarism of the android application. The
original methods to software homology detection is no longer applicable in some cases. In order to fight against new plagiarism technology, we proposed a novel detect method of Android applications’ similarity based on the author’s programming style.

The contributions of this paper are as follows.

(1) A scheme for extracting a series of basic features that can represent the programming style of author.

(2) In order to improve the efficiency of the operation and eliminate the overlap between the features, we tried ICA, PCA and LDA to reduce the extracted basic features. The result shows that features are more effective after dimensionality reduction.

(3) An ELM neural network for software homology detection is introduced; it achieved an accuracy of 88.9% in our experiments.

The rest of this paper is organized as follows. Section 2 reviews the related work for software homology detection. In Section 3, we introduce the proposed method. In Section 4, we introduce the Extreme Learning Machine. Section 5 presents the result of experiments. Moreover, Section 6 concludes the paper.

2. Related work

Software homology detection can be divided into two categories: static methods and dynamic methods. Static methods extract features from source code or assembly code. Static methods can achieve high accuracy, but its accuracy is easily affected by obfuscation. The dynamic method monitors the behavior of applications, which can effectively avoid the interference of confusion, but its temporal and spatial overhead are too high.

So far, many techniques of software homology detection have been proposed. There are four main types: string-based software homology detection, token-based software homology detection, graph-based software homology detection, and syntax-tree-based software homology detection.

The software homology detection that bases on string alignment was firstly proposed by Ottenstein in 1976 [4]. Because it is a comparison of the similarity of the program at the text layer, it can be applied to almost any programming language. And because the analysis of the code structure is omitted, the implementation of the method is simple and the efficiency is high. However, the disadvantages are also obvious. As long as the source code is slightly modified, such as adding some white space, can lead to a sharp decline of the similarity of the source code. There are some mature software based on this method, such as UltraCompare, Beyond Compare and so on.

The principle of token-based software homology detection is that the language program is composed of token sequences at the lexical level. If the two programs have similar token sequences, then the two programs can be considered similar. Among software that base on this method, CP-Miner [5] and CCFinder that developed by Toshihiro Kamiya et al. [6] are relatively matured. The CCFinder firstly performs a certain level of pre-processing, and then compares the intermediate data obtained by these processes. Compared to string-based software homology comparisons, token-based detection can better detect plagiarism that changes the format of origin program.

The graph-based software homology comparison mainly constructs the Program Dependence Graph (PDG) by analysing the syntactic structure, the function call relationship, the control dependency and the data flow of the source code. The similarity of the code is then determined by matching the nodes in the PDG. However, some static analysis tools, which may be time consuming, need to be used in the construction process of PDG, causing this method less efficient.

The syntax-tree-based software homology detection builds an abstract syntax tree (AST) by analyzing the source code and calculates the similarity between these ASTs. The syntax tree-based method includes the structure information of program, so it is very effective to the plagiarism methods that do not change the program’s structure. But its efficiency is influenced greatly by software scale.
3. Proposed method

3.1. Overall architecture

The method we proposed is aim to detect whether the application plagiarizes other applications. The structure of our method is shown in Figure 1, and the core steps of our method are as follows:

1. Collecting Android application packages (APKs) with the author tag as the sample set.
2. Decompiling APK and extracting basic features from configuration files, resource files and source code files.
3. Compressing and optimizing the extracted basic features by using ICA, PCA or LDA.
4. Training the model using ELM with sequences of features vectors and testing the model with reserved data sets.

![Figure 1. The overall architecture of our proposed method](image)

3.2. Extraction of basic features

In this paper, we extract features that reflect the authors’ programming style. The authors’ programming style is the same as the authors’ writing style. It reflects the authors’ programming habits, not the functionality and architecture of the software.

To more fully describe the authors’ programming style, we not only extract features from the source code, but also from the APK file. So the extracted features are divided into two parts. The first part is the information from APK file, including configuration information, path information and so on. Another part includes features that are extracted from the Android source code. We detail some of the most important features in the Section 3.2.1 and Section 3.2.2. Moreover, we list all the extracted features in the table 2.

3.2.1. Features extracted from APK

When an android program is released, all the data and resource files will be packaged into an APK file, which is a compressed file with the suffix .apk. APK file contains all the contents of an Android application, which is the file used by the Android platform to install the application. The APK file is a zip archive. After extracting this package, we can see the structure that is shown in Figure 2. We can find many useful information about the Android application from its APK file. The following are couple of examples.
Apk file

| Directory          | Description                                           |
|--------------------|-------------------------------------------------------|
| Assets directory   | Store static files                                    |
| Lib directory      | Native library on which the program depends           |
| Res directory      | Store source files                                    |
| META-INF directory | Store the signature and certificate of the Application |
| AndroidManifest.xml| Configuration files                                   |
| Classes.dex        | Dex executable file                                   |
| Resources.arsc     | Resource configuration files                          |

**Figure 2.** The structure of the APK file

- **Package path**
  We can get the package name of an application from the AndroidManifest.xml file. For example, the package name of qqlite is “com.tencent.qqlite”. We use PATH1 to record the maximum depth of package path and PATH2 to tag whether the package path make sense. Therefore, the package name above make PATH1 equals to three and PATH2 equals to one.

- **External references**
  In an Android application, lots of constants are used, such as constants that represent colors, string constants, and so on. There are two main methods for using constants. One is to use constants directly. The other is to define constants in the resource file in advance, and then refer to them by @ or # symbol when needed. For example, we can use “@string/app_name” to refer a string constant from the “string” folder. We will use EX to represent the proportion of external references through @ or # symbol to all constants.

### 3.2.2. Features extracted from source code

- **Naming habits**
  During the development of an application, there are many identifiers to be named, such as function name, class name, Activity name and so on. The naming habits in programming are similar to the handwriting habits. Every developer has its own naming habits while different people have different handwriting habits. We conclude author’s naming habits and find the following pattern: double hump, capitalize the first letter, including underline, using the first letter of each word, meaningless single letter, containing uppercase letters, all uppercase, containing digitals, all letters, meaningful, the total length is more than eight. The specific description is showed in the following table (Table 1):

| Style | Example | Description                  |
|-------|---------|------------------------------|
| DH    | BinSh   | Double hump                  |
| CA    | Binsh   | Capitalize the first letter   |
| UL    | bin_sh  | Underline                    |
| SC    | bs      | The first letter of each word|
| SL    | a       | Meaningless single letter     |
| CU    | binSh   | Containing uppercase letters  |
| AU    | BINS    | All uppercase                |
| CD    | bin1    | Containing digitals          |
| AL    | abc     | All letters                  |
| M     | bind    | Meaningful                   |
| L     | binshaaa| The total length is more than eight|

For an identifier, its naming habit is defined as

\[
\text{Code1} = (DH, CA, UL, SC, SL, CU, AU, CD, AL, M, L)
\] (1)
Optional functions
To implement the same functionality, a programmer can use different statements. For example, when implementing a loop function, we can use a “while” statement, a “for” statement, or a “do-while” statement. Similar statements include “if-else” and “switch-case”. Let Code2 be the proportion of the while statement in all loop statements, and Code3 as the proportion of the “if” statement in all conditional statements.

Table 2. Basic features

| Features | Description |
|----------|-------------|
| IMG1     | Naming style of picture. |
| IMG2     | Image format. |
| PATH1    | Maximum depth of package path. |
| PATH2    | Identifies whether the package path makes sense. |
| STR      | The proportion of people names, place names and religious language in the APK file. |
| EX       | The proportion of external references through @ or # symbol to all constants. |
| Per      | Permissions that the Android application applies (such as camera, SMS, phone and so on). |
| Code1    | Naming style of identifier in source code. |
| Code2    | The proportion of the while statement to all loops. |
| Code3    | The proportion of the “if” statement to all loops. |
| Code4    | The ratio of the number of comment lines to the number of lines of code. |
| Code5    | The ratio of the number of line comment lines to the number of all comment lines. |
| Code6    | The ratio of the number of comment lines that immediately follow the code to the number of all comment lines. |
| Code7    | The average length of comment lines. |
| Code8    | The proportion of the left curly braces in a new line to all left curly braces. |
| Code9    | Maximum nesting depth of parentheses (including ‘(’, ‘)’, ‘[’, ‘]’, ‘{’, ‘}’). |
| Code10   | The ratio of the number of functions to the number of lines of code. |
| Code11   | The proportion of the constant first style in statement to determine whether is equal to a constant. |
| Code12   | The proportion of the private classes to all classes. |
| Code13   | The proportion of the functions that have never been used to all functions. |
| Code14   | The proportion of the global variables to all variables. |
| Code15   | The proportion of the static variables to all variables. |

3.3. Feature extraction and dimensionality reduction
Feature extraction converts data patterns to features, which are condensed representations of patterns and contain only salient information [7]. The converted features should retain the maximum amount of information and get the best classification. Principal Components Analysis (PCA) is a technique that finds underlying variables (known as principal components) that best differentiate your data points. It is a linear mapping, which uses the eigenvectors with the largest eigenvalues. However, PCA has some limitations: maximizing spread, interpreting components, orthogonal components. The orthogonal components mean that the principal components generated by PAC must not to overlap in space. To resolve this, we can use an alternative method called Independent Component Analysis (ICA). ICA originated from blind signal separation (BSS) [8]. Compared with Principal components analysis (PCA), ICA utilizes high-order statistical information, which is more conducive to the decomposition of observed signals. Furthermore, ICA allows its components to overlap in space, so they do not need to be orthogonal. Instead, ICA prohibits its components from overlapping in the information it contains to reduce mutual information shared between components. Therefore, ICA may be a backup
technique to use if we suspect that components need to be derived based on information beyond data spread, or that components may not be orthogonal [9]. Linear Discriminant Analysis (LDA) is a supervised linear mapping based on feature vectors, and its classification performance is usually better than PCA.

4. Extreme learning machine
The Extreme Learning Machine (ELM) is an easy-to-use, efficient single-layer feedforward neural network (SLFNS) learning algorithm. In 2004, Associate Professor Huang Guangbin of Nanyang Technological University proposed it. Traditional neural network learning algorithms (such as BP algorithm) need to set manually a large number of network training parameters, and it is easy to generate local optimal solutions. The extreme learning machine only needs to set the number of hidden layer nodes of the network. During the execution of the algorithm, it is not necessary to adjust the input weight of the network and the offset of the hidden nodes, and generate a unique optimal solution, so the learning speed is fast and general. The structure of the ELM is given in Figure 3 [10].

Figure 3. Architecture of Single Layer Feed-forward Network in Extreme Learning Machine Model

For a single hidden layer neural network, suppose there are N arbitrary samples \((X_i, t_i)\), where \(X_i = [x_{i1}, x_{i2}, ..., x_{in}]^T \in \mathbb{R}^n, t_i = [x_{i1}, x_{i2}, ..., x_{im}]^T \in \mathbb{R}^m\).

A single hidden layer neural network with L hidden layer nodes can be expressed as

\[
\Sigma_{i=1}^{L} \beta_i g(W_i \cdot X_j + b_i) = o_j, j = 1, ..., N
\]

where \(g(x)\) is the activation function, \(W_i = [w_{i1}, w_{i2}, ..., w_{in}]^T\) is the input weight, \(\beta_i\) is the output weight, \(b_i\) is the offset of the i-th hidden layer unit. \(W_i \cdot X_j\) represents the inner product of \(W_i\) and \(X_j\).

The goal of single hidden layer neural network learning is to minimize the error of the output, which can be expressed as

\[
\Sigma_{i=1}^{L} ||o_j - t_j|| = 0
\]

That is, to find the appropriate value of \(\beta_i, W_i\) and \(b_i\) that satisfies the following equation.

\[
\Sigma_{i=1}^{L} \beta_i g(W_i \cdot X_j + b_i) = t_j, j = 1, ..., N
\]

Expressed as a matrix,

\[
H\beta = T
\]

\(H\) is the output of hidden layer nodes, \(\beta\) is the output weight, \(T\) is the expect output.

\[
\beta = H^+T
\]
5. Experiment evaluation and result
We conduct our experiment with the game works of six companies crawled from the Internet as the data sets.

5.1. Experimental steps
Our experiment is mainly divided into two parts. The first part is about comparing with other methods of software homology detection. In the second part, we measure the effect of dimensional reduction.

In the first part, we use two methods to confuse the code: variable substitution plus insert independent statement and data stream confusion. And we use totally three methods, including the string alignment, syntax tree analysis and method proposed in this paper, to detect the Android application that has been confused.

In the second part, PCA, ICA and LDA are respectively used to reduce and optimize the basic features. We gradually increase the number of dimension reduction and measure the effect of different descending dimension method.

5.2. Experimental results
Figure 4 shows the accuracy rate of string comparison, grammar tree analysis and our proposed method when detecting the homology of Android applications which are confused by means of variable substitution and insert independent statement. The horizontal axis represents the number of confusing lines of code. Our method achieved the best results, the method of grammar tree analysis is slightly worse and the method of string comparison is the worst.

![Figure 4. Variable Substitution and Insert Independent Statement](image)

In Figure 5, we confuse Android applications with data stream obfuscation. It can be seen from the results that the proposed method is significantly better than the Grammar tree analysis and String comparison methods. When the number of obfuscated lines increases, the advantage is more obvious. For example, when the number of lines of obfuscated code reaches 30 lines, our method still has an accuracy of 75.7%, while Grammar tree analysis and String comparison have only 54.2% and 41.7% accuracy respectively.
Figure 5. Data Stream Confusion

Figure 6 shows the effect of different dimensionality reduction methods on detection accuracy. The accuracy of PCA is almost not reduced when the feature is reduced by 10 dimensions because it has removed redundancy. The performance of ICA is not good. We speculate that the orthogonal components of ICA are not good feature vectors. As the number of dimensionality reductions become larger, the accuracy of both ICA and PCA is significantly reduced, resulting from some useful features are removed. At this case, the advantage of supervised learning is obvious. The accuracy of LDA decreases slowly.

Figure 6. Feature Dimension Reduction

6. Conclusions

In this paper, we propose a novel detect method of Android applications similarity based on the authors’ programming style. We extract a series of basic features that can represent the programming style of author from Android applications. In order to optimize features and improve computational efficiency, we have tried three feature compression techniques. We also build an ELM model with lots of Android applications collected from the Internet. The result of the experiment shows the proposed method performs well when detecting the applications that have been confused by means of variable
substitution, insert independent statement and data stream confusion. Especially in terms of resisting data stream confusion, our approach has obvious advantages.

Reference

[1] China Internet Association. China Internet Development Report 2018[R]. Beijing, 2018.
[2] 360.2015 Android mobile phone application piracy research report[R], 2016.
[3] Rinewalt J D, Elizandro D W, Varnell R C, et al. 'Development and Validation of a Plagiarism Detection Model for the Large Classroom Environment[J]. Computers in Education (CoED), 1986, 6(3): 9-13.
[4] Ottenstein K J. An algorithmic approach to the detection and prevention of plagiarism[J]. Acm Sigese Bulletin, 1976, 8(4):30-41.
[5] Li Z, Lu S, Myagmar S, et al. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code[C]//OSdi. 2004, 4(19): 289-302.
[6] Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code[J]. IEEE Transactions on Software Engineering, 2002, 28(7): 654-670.
[7] Chen V C. Evaluation of Bayes, ICA, PCA and SVM methods for classification[C]//RTO SET symposium on target identification and recognition using RF systems. 2004, 37: 1-12.
[8] Taillard É. Parallel iterative search methods for vehicle routing problems[J]. Networks, 1993, 23(8): 661-673.
[9] Algobeans. (2016). Principal Component Analysis Tutorial. [online] Available at: https://algobeans.com/2016/06/15/principal-component-analysis-tutorial/ [Accessed 22 Nov. 2018].
[10] Huang G B, Zhu Q Y, Siew C K. Extreme learning machine: theory and applications[J]. Neurocomputing, 2006, 70(1-3): 489-50