CAR：Dictionary based Software Forensics Method

Xinyu Yang[1][2]
Beijing University of Posts and Telecommunications, Beijing, 100876, China
E-mail: yangxycl@bupt.edu.cn

Hewei Yu
National Computer Network and Information Security Management Center, Beijing, 100029, China
E-mail: hamutarojp@isc.org.cn

Miao Zhang
Beijing University of Posts and Telecommunications, Beijing, 100876, China
*E-mail:*zhangmiao@bupt.edu.cn

Qi Li
Beijing University of Posts and Telecommunications, Beijing, 100876, China
*E-mail:*liqi2001@bupt.edu.cn

Guoai Xu
Beijing University of Posts and Telecommunications, Beijing, 100876, China
*E-mail:*xga@bupt.edu.cn

With the software inheritance, reuse and piracy becoming more and more common, the software forensics has drawn much more attention increasingly; however, the traditional software similarity detection methods are mainly based on text, grammatical pattern or semantic analyses without deep mining software characteristics. With an attempt to better deal with these problems, this paper proposed the CAR, short for clustering, adjusting and rearranging method as to the software forensics. It focuses on computing the software similarity from the programming characteristics of source codes so as to assist in identifying the software authors by using the clustering algorithm. Experiments proved that whether the comparison of different software versions or the authorship identification of specific software show relativelyidealresults. Consequently, thisproposed method has practical value in the authorship disputes, proof of authorship in court and code re-engineering.

ISCC 2015
18-19, December 2015
Guangzhou, China

[1]This study is supported by National Natural Science Foundation of China Project (61302087, 61401038, U153610079); National High Technology Research and Development Program 863 (2015AA017202)
[2]Speaker and also corresponding author

1. Introduction

Nowadays, the software inheritance, reuse and piracy are being more and more common. Software forensics is of practical significance and application. For example, when a system is attacked and pieces of the source codes as virus or logic bombs are available, tracking the source of such code is of high interest. Other circumstances include the resolution of authorship disputes, proof of authorship in court and code re-engineering. This paper adopts the software programming metrics to compute the similarity on source code level so as to assist in identifying the software authors by using clustering, adjusting and rearranging (CAR) method.

The programming languages, from some aspects, can be treated as a form of language and the writing style of different authors differs greatly. It is believed that each author tends to keep his writing style. Thus, if the source codes are available, it is natural to obtain a great many of features to represent the corresponding authors' programming style; in this sense, an important problem that arises is how to select or extract which subset of numerous performance metrics used for the classification. The metrics as chosen vary significantly, from number or size of classes, methods, loops or variables to include libraries. Researches have been carried out since the initial work of Krsul [1]. They divided the software metrics into three categories, specifically, the programming layout metrics, the programming style metrics and the programming structure metrics. Other research groups select or extract their features in some ways for C and C++ programs. In the paper of Ding et al, these metrics were adapted for java program authorship identification [2]. This paper summarized these advantages and disadvantages of previous works [3-5], and proposed 37 dimension features of author writing style for software consisting of java source files.

Afterwards, this paper also introduces the clustering methods [6-7] into metrics analysis procedure, as comparing similarity only on the programming metrics level decreases the accuracy of software forensics. Multiple source codes in the dictionary display similarity between each other to some extent and exhibit different programming style variations with too much interference information. CAR method can effectively make up for the inadequacy and improve the efficiency and accuracy of experimental results.

The rest of the paper is organized as follows: Section 2 briefly introduces the dictionary based on the software forensics algorithm CAR. Experimental results and analyses are described in Section 3. Finally, conclusions and future work are discussed in Section 4.

2.Proposed Method

2.1Framework Overview

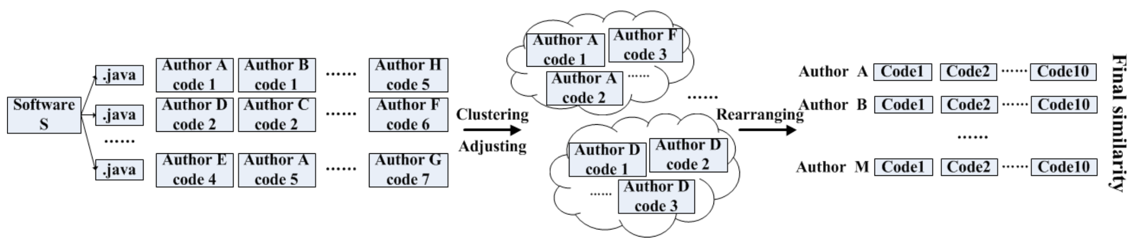

Figure 1: Framework Overview of Proposed Method

Software forensics is a field concerning the evidence of intention from the examination of software. In this paper, it is addressed by computing the similarity of source codes from a dictionary using clustering, adjusting and rearranging, as shown in Fig. 1. Software ***S*** consists of a series of source files, marked as $S_1,S_2,\ldots,S_m$. In order to obtain the ranked list corresponding to S_i (i=1,...,m), each of the source codes with the author labels in the dictionary is compared in a computationally efficient manner. Whereafter, all the source codes marked with the author names are clustered on account of the similarity from certain writing habit and the reliability of each cluster is calculated. The similarity scores of source codes within a cluster are adjusted based on the reliability of that cluster. Finally, the similarity score of each source code is acquired by taking the average of similarity scores of all the source codes belonging to the same author so that we can get the final similarity of each author. Rearrange according to the final similarity from high to low and it is possible that we will find out the author of software ***S***.

2.2 Similarity Measurement Method for Source Code

2.2.1 Extracted Metrics

As the software programming languages in some ways can be regarded as function text, it is possible to measure the program writing style from program layout, naming rules, expression and basic statements, function design these text aspects. The metrics used for software forensics differ in thousands of ways. In reference to previous works and java secure coding standard, we summarize that these metrics can be classified into indention, annotation, naming convention, the choice of loop and condition statements, java class and interface percentage as shown in Table 1. It is worth mentioning that too high dimension could give rise to the curse of dimensionality and may also exert an effect on the efficiency and accuracy of subsequent software similarity judgment model. The proposed 37 metrics avoid computational complexity, which is also enough to cover program writing style on source code level.

The style of list indentation	
	The proportion of center-left curly braces in a line
	The proportion of center-left braces as the first character in a line
	The proportion of center-left braces as the last character in a line
	The proportion of center-right curly braces in a line
	The proportion of center-right braces as the first character in a line
	The proportion of center-right braces as the last character in a line
	The average number of indentation spaces after center-left curly braces
	The average number of tab key after center-left curly braces
Annotation style	
	The proportion of comment lines
	The percentage of / / comments in / / and / * comments
The percentage of conditional statements	
The average spacing distanced to operator left	
The average spacing distanced to operator right	
The percentage of null strings	
The percentage of comments for no comment lines	
The average length of lines	
The length of naming	
	The average length of variable naming
	The average length of function naming

The naming convention of uppercases, lowercases, underscores or dollar signs
The percentage of uppercases
The percentage of lowercases
The percentage of underscores
The percentage of dollar signs
The recycling convention of while, for and do
The percentage of while in loop statements
The percentage of for in loop statements
The percentage of do in loop statements
The percentage of if-else and switch-case
The percentage of if-else accounted for if-else and switch-case
The percentage of switch-case accounted for if-else and switch-case
The percentage of if accounted for if-else
The percentage of switch accounted for switch-case
The percentage of no comment lines in each class/interface
The average number of variables in each class/interface
The average number of methods in each class/interface
The percentage of interface for class
The percentage of variables accounted for not comment lines
The percentage of methods accounted for not comment lines
The percentage of a series of key words accounted for not comment lines ('static', 'try' used in this paper)

Table1: Extracted Metrics

2.2.2 Similarity Measurement Method

As the extracted 37 d metrics are digital form, the Euclidean distance is enough to be used for computing the software similarity. Of course, the smaller the Euclidean distance is, the more similar the compared source codes will be. The calculation method refers to Formula (2.1), where *f* represents 37 d metrics of the question java source file, *f'* stands for the searched one in the dictionary.

$$DIS=\sqrt{(f_1-f'_1)^2+(f_2-f'_2)^2+\ldots+(f_{37}-f'_{37})^2} \qquad (2.1)$$

2.3 Clustering, Adjusting and Rearranging

2.3.1 Computing Ranked Lists

To generate a ranked list $\boldsymbol{R}_i$ corresponding to the input source file S_i, retrieved dictionary source codes are positioned based on their similarity to S_i with the most similar source code positioned at the top of the list. Thus, after obtaining all rank list $\boldsymbol{R}_i$ (i=1,...,m) , a set of ranked lists {$\boldsymbol{R}_1$, $\boldsymbol{R}_2$,...,$\boldsymbol{R}_m$ } are retrieved for software $\boldsymbol{S}$.

2.3.2 CAR Method

Multiple ranked lists of software ***S*** still contain redundant information and it makes the software comparison complicated; therefore, we combine them to generate the author's final similarity ranking from high to low. The calculation steps are elaborated briefly as follows: First, all the source codes across multiple ranked lists are partitioned into various clusters by using *k*-means and the reliability of each cluster is computed as the weighted sum of similarities between the cluster and other clusters across multiple ranked lists; secondly, on the basis of that reliability, the similarity score of every source code in each cluster is adjusted; finally, we obtain the final similarity score and use it to rearrange the similarity list of all authors. The CAR method is shown as Table 2.

Algorithm 1: Compute the ranked list with Clustering, Adjusting and rearranging	
Input	A set of ranked list $\boldsymbol{R}_1,\boldsymbol{R}_2,\ldots,\boldsymbol{R}_{\boldsymbol{m}}$ from multiple source files in software $\boldsymbol{S}$.
Iterate	i=1 to number of source codes in top 10 rank lists.
Clustering	Partition ranked list $\boldsymbol{R}_i$ into different clusters $C_1,C_2,\ldots,C_k$, where k is the number of clusters.
End iterate.	
Iterate	i=1 to number of source files in a cluster, j=1 to k.
Reliability	Compute reliability of cluster $r(C_{i,j})$.
Adjusting	Adjust the similarity score of each source codes from multiple ranked lists according to the reliability calculated above.
End iterate.	
Rearranging	Take the average for all the similarity scores belonging to the same author to compute the final similarity.
Output	Final rank list R' for software $\boldsymbol{S}$.

Table 2 : The Algorithm of CAR Method

a) **Clustering**: programmers tend to refer to others' packaged code segments when the function requirements are relatively independent integrity. Sometimes, they will even copy and paste the same immediately for the sake of convenience and speed; at meanwhile, the programming style of students taught by the same professors or company employees are required to use the same programming specification appears to be similar to some extent; therefore, it is likely that only comparing java file similarity will misjudge the author of question software as someone whose programming style is similar. In the end, this paper adopts the cluster methods. The main idea behind clustering is to congregate source codes in ranked lists where each cluster represents an author programming style profile. In this paper, *k*-means clustering is used as it is computationally faster and produces tighter clusters than the hierarchical clustering techniques. The implementation process of *k*-means algorithm is explained as follows:

Step 1: select k data samples (this paper, in reference to the source code) randomly to serve as the initial cluster centers $c_1,c_2,\ldots,c_k$ of k clusters $C_1,C_2,...,C_k$.

Step 2 : as to each data sample x_p:

Calculate the distance between x_p and each cluster to find out its nearest cluster $C_{nearest}$. In other words, we compute dis(x_p, c_q) in the feature space, thereinto c_q is a certain cluster center, q=1,2,…,k.

Assign x_p to the nearest cluster $C_{nearest}$ and recalculate the cluster center of $C_{nearest}$.

Step 3: Repeat Step 2 until the k cluster centers $c_1,c_2,\ldots c_k$ change no more with judgment standards indicating the square error. *k*-means algorithm aims at minimizing an objective function. In this paper, it can be written as sum of the squared error (*SSE*), as shown in Formula (2.2):

$$SSE=\sum_{q=1}^{k}\sum_{x\in C_q} DIS(c_q,x)^2 \tag{2.2}$$

b) **Adjusting**: on the one hand, it is believed that if the similarity of a source code in a cluster is high, the similarity score of the other source codes in the same cluster should be adjusted to a higher degree; on the other hand, the clusters across multiple ranked lists overlap in terms of common dictionary source codes. The higher the overlap between clusters is, the more likely that they contain source codes with similar features; therefore the reliability of each cluster is computed as the weighted sum of similarities between the cluster and other clusters across multiple ranked lists. The calculation method is elaborated as follows:

$$r(C_{g,r})=\frac{DIS(s_r,C_g)*\sum_{h=1,h\neq g}^{k} DIS(C_g,C_h)}{(k-1)} \tag{2.3}$$

$$DIS(s_r,C_g)=\frac{\sum_{t\in C_g}\|s_r-t\|^2}{|C_g|} \tag{2.4}$$

$$DIS(C_g,C_h)=|C_g\cap C_h| \tag{2.5}$$

The similarity score of each source code in the adjustment stage is illustrated in formula (2.4), where s_r is a certain source code in cluster C_g, and t is the others in the same cluster. $\|s_r-t\|^2$ represents the similarity between the s_r and the other source codes in one cluster by using the Euclidean distance in their characteristic space. The similarity degree between various clusters depends on the number of sharing files computed in Formula (2.5). Finally, on the basis of the above two formulas, the readjusted reliability score of source code s_r in cluster C_g is obtained in Formula (2.3) used for subsequent rearranging.

c)**Rearranging**: after clustering and adjusting, the readjusted reliability score of each source code has been determined already. It is required that the readjusted reliability score of all the source codes belonging to an author should be leveraged to calculate the final similarity. In this paper, we just take the average, that is to say, the final similarity score of an author is the similarity average score of his all selected java files. After that, acquire the similarity of all authors and the author similarity ranking will be rearranged from high to low. It is quite likely to determine the most possible author or team.

3. Experimental Results

3.1Build Dictionary

The dictionary is a large collection of source codes where every author has multiple source files with different styles. Here, we collect the source codes from either academic or freelance sources for java language, all of which are labeled with corresponding author names. In the preprocessing phase, we use 37 regular expressions to build the dictionary. To be specific, process each source code line by line and map it to a 37 d vector for subsequent similarity comparison. It is certain that to make the software forensics more accurate, the dictionary needs to be large enough; but at present, we are still at the experimental stage and the dictionary is not large enough, only containing 1592 files in total.

3.2 Comparison of Different Software Versions

In order to reduce the cost of software development and maintenance, the reusability of the codes between different versions is usually common. Paying attention to the version difference can not only reduce the software testers' workload, but also make contribution to the research work of software forensics. In this section, we put one version into software folder and another version into dictionary to conduct the comparison of different software versions. So the comparison of different software versions is addressed by computing the similarity score of specific software by using a dictionary with its diverse versions.

3.3 Authorship Identification of Specific Software

For the detected software, first of all, the proposed algorithm traverses through the folder to find out all the java files and match the regular expressions to obtain a two-dimensional array, of which the rows are the number of software java files, and the columns are 37; secondly, compute the distance between each java file with all the source codes in the dictionary in turn according to Formula (2.1); thirdly, we can sort out the similarity scores of all the source files belonging to the same author. In order to access the final similarity effectively and efficiently, this algorithm cuts out top 10 most similar source codes in the dictionary.; finally, take the average to acquire the final similarity of each author and rank from low to high. It is worth mentioning that the smaller the final similarity is, the more possible the author of software will be. Experts analyze the results to determine the authorship of specific software.

3.4 Results and Analysis

1) Effects: in this experiment, we download open source codes from Github. We make use of Jedis 2.5 and Jedis 2.6 to conduct software source files detection. Jedis is the Java client of Redis, often used to write Java code to access Redis service in the process of java programming. We put Jedis 2.6 into the software folder while the Jedis 2.5 into the dictionary. Results show the similarity score of top ten source codes as Fig. 2, yet we simply list seven source files of software ***S***.

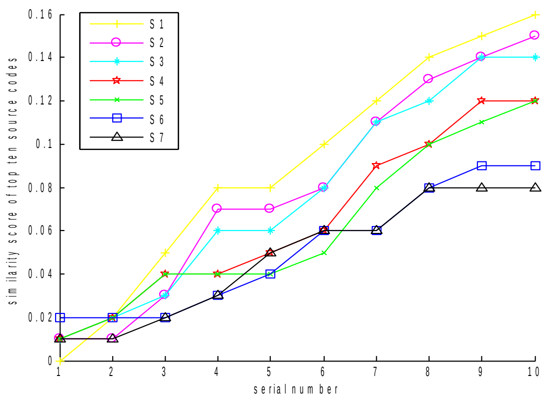

Figure 2: Similarity Score of Top Ten Source Codes Corresponding S_i

As we can see in Fig. 2, the similarity score of the most similar source code of S_1 is zero, indicating that source code is exactly the same as S_1. The similarity score of S_6 and S_7 is lower than the others, indicatingthat the programming style of source codes in the dictionary is more consistent with S_6 and S_7. The curves are beneficial to observe the overall similarity trend.

2)Accuracy: In this section, partial source files of Jedis 2.6 are leveraged to calculate the accuracy, for the sake of brevity and readability, just three source codes results are listed below in Table 3. As AdvancedJedisCommands.java, ClusterCommands.java and JedisCommands.java are all written by a team. It is no wonder that the similar authors is the subset of volunteer A, B, C, D and Jedis 2.5 (The authors' names have been hidden). Although Jedis 2.5 is not the most similar author, it is still retrieved primarily, assisting to find out the actual author manually. The calculation error is within a tolerance range.

1. AdvancedJedisCommands.java		**2. ClusterCommands.java**		**3. JedisCommands.java**	
Author name	**Similarity score**	**Author name**	**Similarity score**	**Author name**	**Similarity score**
Volunteer A	0.01	Volunteer A	0.01	Volunteer A	0.01
Volunteer B	0.01	Volunteer C	0.04	Volunteer B	0.01
Volunteer C	0.04	Jedis 2.5	0.1	Volunteer C	0.03
Jedis 2.5	0.1	Volunteer D	0.14	Jedis 2.5	0.09
Volunteer D	0.14	-	-	Volunteer D	0.13

Table 3: Similarity Degree of Three Java Files

4. Conclusion and Future Researches

In this paper, we combine software feature extracting and clustering method to propose a method for software forensics based on dictionary. The experiment obtains relatively ideal results. In the future, we will expand our dictionary to detect efficiency and acuracy of the proposed method with increase of the dictionary scale.

References

[1] I. Krsul, E.H. Spafford. *Authorship analysis: Identifying the author of a program*[J]. Computer and Security. 16(3):233-257(1997).

[2] H.B. Ding, M.H. Samadzadeh. *Extraction of Java program fingerprints for software authorship identification*[J]. Journal of Systems and Software. 72(1):49–57(2004).

[3] S. Burrows, A.L. Uitdenbogerd, A. Turpin. *Comparing techniques for authorship attribution of source code*[J]. Software: Practice and Experience. 44(1):1-32(2014).

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

[5] A. Desnos. *Android: Static Analysis Using Similarity Distance*[C]. 2012 45th Hawaii International Conference on System Sciences. IEEE 5394-5403(2012).

[6] L. He, L.D. Wu, Y.C. Cai. *Survey of Clustering Algorithms in Data Mining*[J]. Application Research of Computers. 2007(1):10-13(2007)(in Chinese).

[7] Y. X. Luo, D.Y. Fang. *Feature Selection for Software Birthmark Based on Cluster Analysis*[J]. ACTA ELECTRONICA SINICA. 41(12): 2-3(2013).