Vol 8. No. 4, December, 2015
ISSN 2006-1781

African Journal of Computing & ICT

Towards The Development of a Case-Based Reasoning Framework For Software Forensics Analysis

A. Ibitola
Department of Computer Science
Lead City Univerity

A.A. Adigun
Department of Information and Communication Technology,
Osun State University, Osogbo, Nigeria
Email: fempej2013@gmail.com

E.O. Asani
Department of Computer Science
Landmark U niversity
Omu-Aran, Nigeria
Asani.emmanuel@lmu.edu.ng

O.B. Longe
Department of Computer Science & Mathematics
Adeleke University
Ede, State of Osun, Nigeria
longeolumide@fulbrightmail.org

ABSTRACT

We propose the use of a machine learning algorithm for software forensic analysis using case-based reasoning. A reviewof literature was carried out with the objective of identify state of the art in the domain and srtting a research agenda. In this paper, we present preliminary research direction.

Keywords: Case-Based Reasoning, Framework, Software Forensics and Analysis.

African Journal of Computing & ICT Reference Format:
A. Ibitola, A.A. Adigun, E.O. Asani & O.B. Longe (2015): Towards The Development Of A Case-Based Reasoning Framework For Software Forensics Analysis. Afr J. of Comp & ICTs. Vol 8, No. 3. Pp 185-189

1. INTRODUCTION

Software forensics is the use of authorship analysis techniques to analyse computer programs for a legal or official purpose. This generally consists of plagiarism detection and malicious code analysis Software forensics models can be used for identification, classification, characterization and intent analysis. This thesis will concentrate on new advances in software for quantitative data analysis used in forensic authorship identification by examining a selected sample of the art tools. The frequency and severity of the many forms of computer-based attacks such as viruses and worms, logic bombs, Trojan horses, computer fraud and plagiarism of software code (both object and source) have all become increasingly prevalent and costly for many organizations and individuals involved with information systems.

Part of the difficulty experienced in collecting evidence regarding the attack or theft in such situation has been the definition of appropriate measurements to use in models of authorship and the development of appropriate models from these metrics. Source code is the textual form of a computer program that is written by a computer programmer in a computer programming language. These programming languages can in some respects be treated as a form of language from a linguistic perspective or more precisely as a series of languages of particular types, but within some common family. In the same manner that written text can be analysed for evidence of authorship (such as [Sallis, 1994]), computer programs can also be examined from a forensics or linguistics viewpoint [Sallis, et al, 1996] for information regarding the program's authorship.

The figure below shows two small code fragments that where written in a popular programming language called C++ by two separate programme's both programs provide the same functionality (calculating the mathematical function factorial (n), normally written as n!) from the users perspective that is to say, the same inputs will generate the same outputs for each of these programs.

```
//  factorial takes an integer as an input and returns
// the factorial of the input
// This routine does not deal with negative values!

Int factorial ( int input )
{
            int counter;
            int fact;
            fact = 1; // initalizies fact to 1 since factorial 0 is 1
            for ( counter = input; counter > 1; counters =
counter -1)
            {
                        Fact = fact * counter;
}

int f ( int x ) {
int a, y = 1;
if ( !x) return 1; else return x * f ( x – 1);}
```

1.1 Program Segments in C++

As should be apparent each programmer has solved the same problem in both a different manner (algorithm) and with a different style exhibited in his/her code. The first algorithm is a simple loop through the values from 1 through to the input into the function (in reverse), while the second employs a more sophisticated (but also worse performing) recursive definition. The stylistic differences include the use of comments, variable names, use of white space, indentation and the levels of readability in each function.

These fragments are obviously far too short to make any substantial claims. However, they do illustrate the ability for programmers to write programs in a significantly different manner to another programmer without any instruction to do so. Both of these functions were written in the natural styles of their respective authors and so should reflect the types of differences that should be evident in general between their programs

2. LITERATURE REVIEW

The general methodology of authorship attribution applies to both natural and computing languages. Although source code is much more grammatically restrictive than natural languages, there is still a large degree of flexibility when writing a program (Krsul and Spafford 1996). Computational authorship attribution methodology for both natural and computing languages requires two main steps (Krsul and Spafford 1995; Chaski 1997, 2005; MacDonell and Gray 2001, Ding and Samadzadeh 2004). The first step is the extraction of variables representing the author's style. Ideally, authorial features should have low within-author variability, and high between-author variability (Krsul and Spafford 1996, Kilgour, Gray, Sallis and MacDonell 1997, Chaski 1997).

The second step is applying a statistical or machine learning algorithm to these variables in order to develop models that are capable of discriminating between several authors. Defining the variables and discovering the best classification algorithm for the defined variables is difficult, empirical task, but it is feasible and prevents subjective pronouncements which are no longer considered by courts to be acceptable scientific forensic evidence (Chaski 1997, 2005).

Authorship Attribution Methods for Computing Languages
In general, when authorship attribution methods have been developed for computing languages, the suggested software features are programming language-dependent and require either computational cost or hand-coding for their calculation. The main focus of the previous work was the definition of the most appropriate features for representing the style of an author (Oman and Cook 1989; Longstaff and Shultz 1993; Spafford and Weeber 1993; www.ijde.org 2International Journal of Digital Evidence Spring 2007, Volume 6, Issue 1

Sallis, et. al. 1996). For author identification in computing languages, proposed metrics have included, for example, indentation, placement of comments, placement of braces, character preferences, construct preferences, statistical distribution of variable lengths and function name lengths, statistical distribution of lines of code per function, ratio of keywords per lines of code, spelling errors, the degree to which code and comments match, and whether identifiers used are meaningful. This list shows that many of the previously proposed features either cannot be measured objectively in any source code program (a condition which also plagued natural language authorship identification methods, until very recently) or require hand-coding.

Krsul and Spafford (1995) developed a software analyzer program to automate the coding of software metrics. The software analyzer extracted layout, style and structure features from 88 C programs belonging to 29 known authors. A tool was developed to visualize the metrics collected and help select those metrics that exhibited little within-author variation, but large between-author variation. Discriminant function analysis was applied on the chosen subset of metrics to classify the programs by author. The experiment achieved 73% overall accuracy.

MacDonell and his colleagues (Kilgour, Gray, Sallis and MacDonell 1997; Gray, Sallis and MacDonell 1998; MacDonell and Gray 2001) have automated authorship identification of computer programs written in C++. Gray, Sallis and MacDonell 1998 developed a dictionary-based system called IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination) to extract source code metrics for authorship analysis. In MacDonell and Gray's 2001 work, satisfactory results were obtained for C++ programs using case-based reasoning, feed-forward neural network, and multiple discriminant analysis. The best prediction accuracy – at 88% for 7 different authors--- was achieved using Case-Based Reasoning.

Focusing on Java source code, Ding and Samadzadeh (2004) investigated the extraction of a set of software metrics that could be used to identify the author. A set of 56 metrics of Java programs was proposed for authorship analysis. The contributions of the selected metrics to authorship identification were measured by canonical discriminant analysis. Forty-six groups of programs were diversely collected. They achieved a classification accuracy of 87.0% with the use of canonical variates.

This brief review of previous work reveals four criteria for our own research agenda. First, metrics selection is not a trivial process and usually involves setting thresholds to eliminate those metrics that contribute little to the classification model. Second, some of the metrics are not readily extracted automatically because they involve subjective judgments. Third, many software metrics are programming-language dependent. For example, metrics useful for Java programs cannot be used for examining C or Pascal programs. Fourth, even with automated feature extraction and analysis, the classification accuracy rates do not reach 90%.

In sum, the previous work in author identification of programming code has suffered from language-dependence, manual coding of subjective features and accuracy rates below 90%. In this context, our goal is to provide a fully-automated, language-independent method with high reliability for distinguishing authors and assigning programs to programmers.

3. PROBLEM STATEMENT

It seems clear that there are many potential factors that could be examined to determine authorship of a piece of software. Ideally, this analysis would be used to identify a suspect and then search would be made of storage and archival media to locate incriminating sources. However, a more likely scenario would see a set of metrics and characteristics derived from the code remnant and then compared with representative samples written by the suspects. This comparison must be made with considerable care, however, to prevent complicating factors from producing either false positive or false negative indications.

One such complication, for instance, is the amount of code compared. A small amount of suspect code (e.g., a computer virus) might not be sufficient to make a reasoned comparison unless very unusual indicators are present. Another complication is the reuse of code. If the author has reused code from her earlier work, or code written by others, the effect may be to skew any metrics derived from the suspect code. It might be enough to correctly indicate *original* authorship, but that might not identify the actual culprit. In some cases, code reuse may be obvious and it may be omitted from the comparison. However, there may be cases where that is not possible. Likewise, if the suspect code was written as part of collaboration, the characteristics of the individual authors may be subsumed or eliminated entirely.

A clever programmer, aware of this method, might disguise his code. This would probably involve using different algorithms and data structures than what he would normally use. Although this might eliminate the possibility of a match based on internal characteristics, it might also make the code more likely to fail in use. This should also make the programmer use more testing, and keep intermediate versions of the program that could later be matched against the suspect code.

There is also the potential that the underlying application may have a strong influence on the overall style and nature of the code. For instance, if we are attempting to match characteristics of a small MS-DOS boot record virus, and the code we compare against is for a UNIX-based screen editor, it is unlikely that we would find much correspondence between the two, even if they were written by the same author. Therefore, we must be certain that we compare similar bodies of code.

4. AIM AND OBJECTIVES

The aim of this thesis is to build some modules for analysing the resultant metric data including case based reasoning. The specific objectives of this research are as follows:

a) Identify the limitations or drawbacks of existing software forensics aids in the determination of malicious code authorship;
b) Propose an enhanced software architecture model based on the limitations in (a);
c) Simulate and carry out performance evaluation of the enhanced model.

6. RESEARCH METHODOLOGY

Software development methodology can be defined as a framework or approach that is used to structure, plan and control the process of developing a software product or information systems. Therefore, the following method will be employed in ach/ieving the research objectives:

a. Various and relevant extensive review of software architecture frameworks will be carried out from existing literature.

b. Discovery of the state-of-the-earth and state-of-the-practice of existing software architecture frameworks will be analyzed through the aid of questionnaire and interview methods.

c. Analysis of information elicited from (a) and (b) will be executed with the aid of a suitable computational tool.

d. The proposed model will be developed with the Unified Modeling Language (UML) using Agro UML Computer Aided Systems Engineering (CASE) tools and

e. The proposed model will be simulated with DEVSJAVA simulation kit and selected parameters will be used to evaluate the performance of the model.

6. EXPECTED CONTRIBUTION TO KNOWLEDGE

There are many differences between handwritten prose and computer programs. Handwriting samples are usually fixed in an instant and prose is usually not incrementally developed, while a program evolves over time. Multiple changes to a section of code as a program is developed can lead to a structure that the author would have been unlikely to create under other circumstances.

Coding is also different in that code written by others is often incorporated into a program. Often, a program is not the result of the influence of only one author. We suspect that this would severely impair the selection of writer-specific code features without knowledge of the development of the program.

Nonetheless, if there is a sufficiently large sample of code and sufficient suspect code, if there are unusual features present, and if we have correctly chosen our points of comparison, this method may prove to be quite valuable. Currently, similar *ad hoc* methods are used by instructors when they compare student assignments for unauthorized collaboration (cheating). The samples are usually not big, but the characteristics are often distinctive enough to make valid conclusions about authorship. Developing and applying more formal methods should only improve the accuracy of such methods, and make them available for more in-depth investigations.

Not only would a formal method of *software forensics* aid in the determination of malicious code authorship, it would have other uses as well. For instance, determining authorship of code is often central to many lawsuits involving trade secret and patent claims. The characteristics we have outlined in this work might be used to determine if code is, in fact, original with an author or derived from other code. However, a rigorous mathematical approach is needed if any of these kinds of results are to be applied in a court of law.

We believe that if this approach is developed, it may also prove useful in applications of reverse-engineering for reuse and debugging. The analysis of code to determine characteristics is, at the heart, a form of reverse-engineering. Existing techniques, however, have focused more on how to recover specifications and programmer decisions rather than to determine programmer-specific characteristics.

Vol 8. No. 4, December, 2015
ISSN 2006-1781
African Journal of Computing & ICT

Further research into this technique, based on examination of large amounts of code, should provide further insight into the utility of what we have proposed. This work will determine which characteristics of code are most significant, how they vary from programmer to programmer and how best to measure similarities. Different programming languages and systems will be studied to determine environment-specific factors that may influence comparisons. And most importantly, studies should be conducted to determine the accuracy of this method; false negatives can be tolerated, but false positives would indicate that the method is not useful for any but the most obvious of cases.

REFERENCES

Andrew Gray, Philip Sallis, and Stephen MacDonell. "Software Forensics: Extending Authorship Analysis Techniques to Computer Programs." Proceedings of the 3rd Biannual Conference of the International Association of Forensic Linguists (IAFL). Durham NC, USA, 1997

Spafford, E. H., and Weeber, S. A. (1993). "Software forensics: tracking code to its authors." *Computers and Security*, 12:585-595.

Gray, A.R., Sallis, P.J., and MacDonell, S.G. (1998). IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination): A dictionary-based system for extracting source code metrics for software forensics. Submitted to *SE:E&P'98 Software Engineering: Education & Practice*. Dunedin. New Zealand.

Kilgour, R.I., Gray, A.R., Sallis, P.J., and MacDonell, S.G. (1997). A Fuzzy Logic Approach to Computer Software Source Code Authorship Analysis. Accepted for *The Fourth International Conference on Neural Information Processing -- The Annual Conference of the Asian Pacific Neural Network Assembly (ICONIP'97)*.Dunedin. New Zealand.

Longstaff, T.A., and Schultz, E.E. (1993). Beyond Preliminary Analysis of the WANK and OILZ Worms: A Case Study of Malicious Code. *Computers & Security*. 12:61-77.

Sallis, P.J. (1994). Contemporary Computing Methods for the Authorship Characterisation Problem in Computational Linguistics. *New Zealand Journal of Computing*. 5(1):85-95.

Sallis P., Aakjaer, A., and MacDonell, S. (1996). Software Forensics: Old Methods for a New Science. *Proceedings of SE:E&P'96 (Software Engineering: Education and Practice)*. Dunedin. New Zealand. IEEE Computer Society Press. 367-371.

Spafford, E.H. (1989). The Internet Worm Program: An Analysis. *Computer Communications Review*. 19(1):17- 49.

Spafford, E.H., and Weeber, S.A. (1993). Software Forensics: Can we track Code to its Authors*? Computers & Security*. 12:585-595.

Whale, G. (1990). Software Metrics and Plagiarism Detection. *Journal of Systems and Software*. 13:131-138.