Full citation: Sallis, P.J., Aakjaer, A., & MacDonell, S.G. (1996) Software forensics: old methods for a new science, in Proceedings of Software Engineering: Education & Practice (SE:E&P'96). Dunedin, New Zealand, IEEE Computer Society Press, pp.481-485.
doi: 10.1109/SEEP.1996.534037

Software Forensics: old methods for a new science

Philip Sallis, Asbjorn Aakjaer and Stephen G. MacDonell

Computer and Information Science
University of Otago, Dunedin, New Zealand

Abstract

Over the past few years there has been a renewed interest in the science of software authorship identification; this area of research has been termed 'Software Forensics'. This paper examines the range of possible measures that can be used to establish commonality and variance in programmer style, with a view to determining program authorship. It also describes some applications of these techniques, particularly for establishing the originator of programs in cases of security breach, plagiarism and computer fraud.

1. INTRODUCTION

Programming style has fascinated the computing community from the earliest days of computer programming. The stylistic influence of an individual on algorithm implementation within the constraints of a given programming language is limited but can be identified to some extent as traits or tendencies in the expression of logic constructs, data structure definition, variable and constant names and calls to fixed and temporary data sets. To some extent, the stylometric methods used in literary analysis and computational linguistics can be applied to the quest for program authorship identification, particularly in determining the frequency of a determined set of characteristics, such as data name instances. Similarly, software science measures and conventional metrics such as *McCabe's Program Complexity Measure* [12] can be used to compare programs in an attempt to identify authorship commonality.

Forensic analysis as a term refers to 'after the fact' experimentation and study. Software Forensics is the area of Software Science aimed at authorship analysis of computer source code. Authorship analysis of any kind, whether it be applied upon source code or written word, is based upon the premise that authors develop a style and approach that is identifiable. Although there is no formal proof that a computer program has embedded within it the characteristics of the author, one can see merely by looking at two code fragments that each author has their own style, and methods.

Software Forensics therefore endeavours to use observable characteristics to determine authorship of code fragments. Individual authors have their code examined to determine their 'image' or stylistic profile. Program code fragments for analysis are compared with the author profiles. Statistical methods are then used to determine which author is the most likely to have authored the code.

2. WHAT CAN SOFTWARE FORENSICS OFFER?

Software Forensics is a new area of research with to date, limited published results. Previous studies have not defined how accurate the analysis methods are, or what factors may influence the result. It can be seen that the greater the number of possible authors in any sample for analysis, the greater the chance of error.

From a scientific point of view, Software Forensics is a modern attempt at an old science. The metric-based approach, when combined with natural language processing (computational linguistics), suggests an exciting realm for new research.

Since the first reference to Software Forensics [16], its base has been in computer security. When a security breach has been detected, often the only evidence other than the damage, is the occasional code fragment. Software Forensics is an attempt to help determine if two or more fragments were authored by the same person. This could be valuable information, especially if security breaches are repeated. If for instance, the code was written in-house, Software Forensics could help find the culprit.

With the huge expansion of the Internet and other multinational networks, it is important to have some form of software auditing. Software Forensics may well find a place within this domain.

3. USING TRADITIONAL METHODS FOR SOFTWARE FORENSICS RESEARCH

Forensics analysis or authorship analysis has in the past been primarily based on software metrics. Metrics are chosen that distinguish the author-specific portions of the code being analysed. Hence, the choice of metrics is crucial to the eventual result.

This is true of any statistical study, where a variable, or a group of variables determines the outcome. When the variables change, the entire solution changes. This dependence can be decreased by choosing a representative set of metrics. This has the effect of decreasing the weighting of each metric, thereby reducing the possible deviation that any one metric can cause, providing more robust results.

In 1994, Ivan Krsul published a paper discussing the results of a study completed at Purdue University [9]. The study showed that no particular metric or group of metrics demonstrated the ability to identify an author *outright.* This is to be expected, however, as Software Forensics can only be considered a tool; it will never be able to determine beyond all doubt the author of a piece of code.

The Purdue study identified a large range of metrics that can be used to help determine the author of a program. The metrics were collected from a variety of sources [3,8,14,18], with three particular groups of metrics identified:

- Program Layout Metrics
- Program Style Metrics
- Program Structure Metrics

The study produced good results - up to 78% of code fragments were classified correctly. These code fragments however, were collected in a relatively controlled environment (university environment). Unfortunately the study did not give conclusive results as to which type of metric gives best results, nor did it propose a generic set of metrics that could be used in further studies.

The use of code-based metrics in software authorship identification has some promise, but its effective application may be most fruitful in the area of plagiarism detection. In fact, the two objectives of authorship identification and plagiarism detection may produce conflicting evidence for a given pair of programs, in that whilst the two programs may return very similar values for structural metrics (which might suggest plagiarism) it may be that the authors of those programs are very much distinct, and were simply taught by the same instructor. This type of conflict could be overcome, however, by a more comprehensive approach to code analysis, incorporating some of the lexical methods described in the final section of this paper.

3.1. Authorship identification

It is our contention that programmers adopt certain styles; characteristics that are evident (and therefore measurable) in the programs they write. Given that code examples are collected on a regular basis from each programmer, specific profiles can be developed using this common set of characteristics.

Software science metrics [4] and those based on the control-flow of programs (e.g. [12]) produce metric values that are clearly program-specific (see the following subsections for definitions of these metrics). This implies that the identification of the author of a code segment or program using these metrics is dependent on the existence of a functionally similar program in the previously collected sample. Given extensive software reuse by individual developers, this may be quite possible - a programmer is likely to use (and adapt if necessary) a previously developed code segment to execute the same function in different systems. (If reuse is based on a common library, however, with global access by all programmers, identification of an individual author with such metrics is less likely.)

A simplified example may help to illustrate this approach. In a previous project a programmer may have developed a bubble sort routine as part of a file update subsystem. This program has been tested and is error-free. The programmer now finds that a similar routine is needed in her/his current project. Rather than write it from scratch, the programmer is likely to retrieve the routine and adapt it to suit the current application. This may involve changing some data item names and formats, but much of the *structure* (including the control structures used, the control flow through the routine, and the number of program tokens employed) will remain essentially the same. Thus the profile of the structural characteristics will also be the same, and authorship, if unknown, could be established with some confidence in this manner. This confidence could be significantly increased with the complementary use of token frequency analysis in the identification process.

If the intent of a program is malicious then the programmer may make an earnest attempt to disguise the code, through the use of characteristics that are inconsistent with their profile. In these circumstances authorship verification based on traditional software metrics is less likely. For one thing, it is highly improbable that a previous functionally equivalent program would be held in the collected sample! For another, the skills of the programmer in disguising the code could make identification from a purely structural basis virtually impossible. It is under these circumstances, however, that lexical analysis, which is concerned with variable names and layout characteristics (e.g. all variables in lower case), may be used to greater effect.

3.1.1. Halstead's Software Science

The basis of Halstead's theory involves the identification of operands and operators in the expression of algorithms and then applying certain manipulations to the counts of these basic elements to obtain quantitative measures. Halstead defined operands as "...variables or constants" and operators as "...symbols or combinations of symbols that affect the value or ordering of an operand" [4 p.5]. These and other basic properties are denoted as follows:

$n1$ = number of unique or distinct operators

$n2$ = number of unique or distinct operands

$N1$ = total usage of all the operators

$N2$ = total usage of all the operands

The *vocabulary* is derived from these initial figures as:

$n = n1 + n2$

and the implementation *length* as

$N = N1 + N2$

One of the primary measures formulated from the element counts was the size measure, *volume:*

$V = N \log2 n$

This is said to be a reflection of the number of mental comparisons required to generate a program.

3.1.2. McCabe's Cyclomatic Complexity

McCabe's measure [12] uses the number of execution paths through program code as an indication of code complexity, as each path must be traced if the program is to be completely understood. McCabe's metric (v(G)) is derived from a flowgraph representation of program code, where nodes are blocks of sequential statements and edges are processing paths. The metric is calculated in the following way:

$v(G) = e - n + 2p$

where e = the number of edges in the graph

n = the number of nodes in the graph

p = the number of components in the graph

(p = 1 for one module).

A 'short-cut' method of calculation has also been widely promoted, based solely on the number of decision structures (selection and iteration) in the code [5,12]:

$v(G) = \pi + 1$

where π = the number of decisions structures in the code.

3.2. Plagiarism detection

A substantial body of research into the detection of plagiarism in programs using software metrics already exists (for example, see [6,10,17]). This is a specific case of authorship *verification* that is in many ways easier to undertake than strict *identification.* This is clearly due to the fact that in many cases the programs being compared are, by definition, functionally equivalent. Thus there is no need to refer back to collected samples of a programmer's work so as to determine the authorship of a new program.

Of the more recent attempts to develop a set of measures for plagiarism detection, Leach [10] promotes a more comprehensive approach, in that characteristics other than those normally considered are proposed. In order to augment the token and control flow measures, Leach suggests the use of indicators that consider the degree of coupling between modules (through data and control transfer). This coverage helps to ensure that three different aspects of a program are considered in the analysis - volume, control flow and structure - so that changes to one aspect in order to avoid plagiarism detection will not necessarily go unnoticed.

To continue this development, we would suggest that the list of metrics should be increased further, so that a comprehensive characteristic vector of relatively independent indicators can be used as a baseline for program comparison. This would require the inclusion of a data dependency metric, a nesting level metric and a control structure metric. Thus a six-tuple vector should provide ample coverage of the various program structure characteristics that might vary (or remain the same in cases of plagiarism) according to author. Candidate metrics for these three extra categories will need to be tested for their effectiveness, but the following might be included:

Data dependency - Just as control flow dependency is assessed from a directed graph using the McCabe measure, data dependency can be measured from a similar representation. Bieman and Debnath [1] suggest the development of a Generalised Program Graph (GPG), in which nodes denote both predicate clauses *and* variable definitions. Measures derived from such a representation should therefore enable the assessment of both aspects.

Nesting level - Measures for program, module and average nesting depth may be derived from the work of Dunsmore [2]. Computation of the program and module measures involves assigning each line of code a nesting level indicator, as follows:

(i) the first executable statement is assigned a nesting level of *1*

(ii) if statement a is at level *l* and statement b simply follows sequentially the execution of statement a, then the nesting level of statement b is *l* also

(iii) if statement a is at level *l* and statement b is in the range of a loop or a conditional transfer controlled by statement a, then the nesting level of statement b is *l* + 1.

The sum of all the statement levels in the program or module produces the total nesting depth measures. Average nesting depth can be calculated by dividing the total measures by the number of statements in the program/module.

Control structure - This particular set of indicators provides a link to the lexical measures suggested elsewhere in this paper, in that specific structures produce different values for use in the profile. Two examples of such an indicator are the MEBOW measure [7] and the NPATH measure [13]. MEBOW (Measure Based On Weights) assigns various weightings to expression types (e.g. an IF...THEN construct has a weighting of 3) and the sum of the weightings is used to denote program complexity. NPATH adopts a similar approach but is more comprehensive in the constructs considered. The final calculation of NPATH is then based on the product of the individual construct values.

Thus we might reach a point where a profile of characteristics that enables or improves plagiarism detection includes:

Volume: Halstead's *n*, *N* and *V* measures

Control flow: McCabe's v(G) measure

Structure: Leach's coupling assessment

Data dependency: Bieman and Debnath's GPG assessment

Nesting depth: Dunsmore's program nesting depth and average nesting depth measures

Control structure: Nejmeh's NPATH measure

It should be noted here that we are not assessing the value of the metrics for their originally intended purpose, that is, program complexity assessment. This topic has already seen widespread research and publication and is far beyond the scope of this work. Rather it is our assertion that, when combined with the output of lexical analysis, an overall set of characteristics like the one described above should enable effective plagiarism detection and (perhaps to a lesser extent) independent authorship identification.

4. THE NLP APPROACH TO STYLISTICS AND AUTHORSHIP ANALYSIS

Establishing the authorship of documents, letters and other written works has posed a challenge to linguists for hundreds (and probably thousands) of years. Several well-known techniques have been used to characterise authorship such as hand-writing recognition and word usage. More subtle indicators such as author's expressions or turns of phrase, their dependence on certain words or phrases, the frequency of individual words, preference for the use of short or long sentences, prosaic language, and so on, all contribute to a profile or set of individual authorship characteristics [11]. Sadly, research in this area suffers from inadequacies in the methodologies that are employed. Criterion-based analysis techniques are too rigid for the qualitative dimension often required to perform the desired analysis in computational linguistics. These concerns notwithstanding, we are mostly left with document statistics as the only way to satisfactorily distinguish one text from another, and inevitably one author from another.

Statistical techniques are generally employed to discern trends, frequencies and correlations from data gathered out of written text or documents in attempts to establish authorship style. Many examples of the application of statistical methods to this area of analysis exist in contemporary research literature (e.g. see [11]). Data is usually gathered from an analysis of:

1. the average length of sentences (in words)
2. the average length of paragraphs (in sentences)
3. the use of passive voice (expressed as a percentage)
4. the number of prepositions as a % of total words
5. the frequency of ‘function words’ used in each text.

One technique, illustrated in Sallis [15], is to apply a set of ‘function words’ that have been used to establish authorship in another document corpus, to a new set of documents, and to compare the results with a known or validated authorship style. The set of function words is established from a word frequency analysis of a large single-author corpus. The most celebrated of these function word sets has been developed out of an analysis of the Shakespearean Canon. This set can be as large or small as desired but generally includes a basic set of nineteen words such as ***a, and, but, by, for, from, in, it, of, that, the, to, with, I, you, my, me, is, not***. This set is the result of word frequency analysis and is used throughout the literature as a reliable analytical tool. In itself, it is not very useful for application to software but the approach may have some merit. The set contains a sample of personal pronouns, prepositions and verbs.

Readability scores such as the Gunning-Fog index and other literary measures are used for experiments in computational linguistics but only apply to domain-independent and context-free grammars, where written text is being analysed. Again, this does not apply to computer software.

Unfortunately then, this approach has limited value for Software Forensics because there is no similar set of ‘function words’ available. In fact, working in the restricted domain of computer programming languages, a defined set of allowable words exists, which limits the variability of free expression. There may be some future in analysing data and variable names but we are then faced with the problem of obtaining sufficient discrete works of assumed single authorship with a known product.

Studies of samples with observed and expected values often lend themselves to using chi-square methods and binomial distributions, but usually the significance levels are not of sufficient magnitude for confident conclusions to be drawn about the positive identity of authors. Multivariate statistics are also popular, particularly when syntactic elements are also counted in the analysis. Ledger [11] illustrates this in attempting to determine differences in letters within a corpus, where no independent authenticated letter exists. He uses cluster analysis and illustrates his results with scatter grams to provide a spatial dimension to the stylistic differences between text. In keeping with a number of results from this area of research, differences in style across a corpus of documents or letters that are supposed to have been written by a single author, often suggest that the work has been contributed to by others to a greater or lesser extent. They do not usually authenticate absolutely that a single author was responsible, although clearly this may be confounded, for example, by changes in an individual's style. Interesting as this may be, it is unlikely that much can be gained from these approaches *in isolation* in analysing software for authorship identification.

5. CONCLUSIONS

It would seem from an initial review of available techniques that no single approach will adequately suffice for Software Forensics. A combination of techniques from conventional software metrics and computational linguistics appears to be the way ahead and a great deal of experimental work needs to be carried out in order to advance a theory of software authorship identification.

Of prime difficulty in experimental design for this research is the identification of a suitable software sample, even within a domain-dependent context. Next, the establishment of an individual programmer's authorship style that is robust enough for authentication across the sample will require some unique analytical methods in order to maintain validity, even within acceptable statistical norms for probabilistic inference. The creation of individual 'profiles' will require extensive empirical work across a non-trivial set of software. The established profiles will then need to be verified through comparison with a random set of software. In order for any inferences to be meaningful from this work, a large sample of software and authors will be required, highlighting the long-term nature of the project.

REFERENCES

[1] Bieman, J.M. and Debnath, N.C. An analysis of software structure using a generalized program graph. *Proceedings COMPSAC '85* (1985) 254-259

[2] Dunsmore, H.E. Software metrics: an overview of an evolving methodology. *Information Processing & Management* 20 (1984) 183-192

[3] Conte, S., Dunsmore, H.E. and Shen, V. *Software Engineering Metrics and Models.* Benjamin/Cummings Publishing Company, Menlo Park CA (1986)

[4] Halstead, M.H. *Elements of Software Science.* Elsevier North-Holland, New York (1977)

[5] Hansen, W.J. Measurement of program complexity by the pair (cyclomatic number, operator count). *ACM SIGPLan Notices* 13 (1978) 29-33

[6] Jankowitz, H.T. Detecting plagiarism in student Pascal programs. *The Computer Journal* 31 (1988) 1-8

[7] Jayaprakash, S., Lakshmanan, K.B. and Sinha, P.K. MEBOW: a comprehensive measure of control flow complexity. *Proceedings COMPSAC '87* (1987) 238-244

[8] Kernighan, B. and Plauger, P. *The Elements of Programming Style.* McGraw-Hill (2nd ed), New York (1978)

[9] Krsul, I. *Authorship Analysis: Identifying The Author of a Program,* Technical Report CSD-TR-94-030, Department of Computer Sciences, Purdue University (1993)

[10] Leach, R.J. Using metrics to evaluate student programs. *ACM SIGCSE Bulletin* 27 (1995) 41-43, 48

[11] Ledger, G. An exploration of differences in the Pauline epistles using multivariate statistical analysis. *Literary and Linguistic Computing,* 10 (1995) 85-98

[12] McCabe, T.J. A complexity measure. *IEEE Transactions on Software Engineering* 2 (4) (1976) 308-320

[13] Nejmeh, B.A. NPATH: A measure of execution path complexity and its applications. *Communications of the ACM* 31 (1988) 188-200

[14] Oman, P.W. and Cook, C.R. A programming style taxonomy. *Journal of Systems and Software* 15 (1991) 287301

[15] Sallis, P.J. Contemporary computing methods for the authorship characterisation problem in computational linguistics. *New Zealand Journal of Computing* 5(1) (1994) 85-96

[16] Spafford, E.H. and Weeber, S.A. Software forensics: Can we track code to its authors? *Computers & Security* 12 (1993) 585-595

[17] Whale, G. Software metrics and plagiarism detection. *Journal of Systems and Software* 13 (1990) 131-138

[18] van Tassel, D. *Program Style, Design, Efficiency, Debugging, and Testing.* Prentice Hall, Englewood Cliffs NJ (1978)