Authorship Analysis: Identifying The Author of a Program.

Ivan Krsul Eugene H. Spafford*

The *COAST* Project
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{krsul,spaf}@cs.purdue.edu

October 4, 1995

Abstract

Authorship analysis on computer software is a difficult problem. In this paper we explore the classification of programmer's style, and try to find a set of characteristics that remain constant for a significant portion of the programs that this programmer might produce.

Our goal is to show that it is possible to identify the author of a program by examining its programming style characteristics. Ultimately, we would like to find a signature for each individual programmer so that at any given point in time we could identify the author of any program.

The results of this paper support the conclusion that within a closed environment, and for a specific set of programmers, it is possible to identify a particular programmer and the probability of finding two programmers that share exactly those same characteristics should be small.

1 Introduction

There are many occasions in which we would like to identify the source of some piece of software. For example, if after an attack to a system by some software we are presented with a piece of it, we might want to identify its source. Typical examples of such software are Trojan horses, viruses, and logic bombs[1].

Other typical circumstances will require that we trace the source of a program. Proof of code re-engineering, resolution of authorship disputes and proof of authorship in courts are but a few of the more typical examples of such circumstances. Often, tracing the origins of the source requires that we identify the author of the program.

Given that software evolves over time, that programmers vary their programming habits and their choice of programming languages, and that software gets reused, it seems unlikely that, given a piece of software, we will identify the programmer who wrote it out of the millions of programmers who develop software.

However, the identification process in computer software can be made reliable for a subset of the programmers and programs.

2 Statement of the Problem.

Authorship analysis in literature has been widely debated for hundreds of years, and a large body of knowledge has been developed [Dau90]. Authorship analysis on computer software, however, is different and more difficult than in literature.

Several reasons make this problem difficult. Authorship analysis in computer software does not use the same stylistic characteristics as authorship analysis in literature. Furthermore, people reuse code, programs are developed by teams of programmers, and programs can be altered by code formatters and pretty printers.

Our objective is to classify the programmer and to try to find a set of characteristics that remain constant for a

*Contact person for questions concerning the paper.

[1] [GS92] defines trojan horses as programs that appear to have one function but actually perform another function; viruses as programs that modify other programs in a computer, inserting copies of themselves; and logic bombs as hidden features in programs that go off after certain conditions are met.

significant portion of the programs that this programmer might produce. This is analogous to attempting to find characteristics in humans that can be used later to identify a specific person.

Eye and hair coloring, height, weight, name and voice pattern are but a few of the characteristics that we use on a day-to-day basis to identify persons. It is, of course, possible to alter our appearance to match that of another person. Hence, more elaborate identification techniques like fingerprinting, retinal scans and DNA prints are also available, but the cost of gathering and processing this information in large quantities is prohibitively expensive. Similarly, we would like to find the set of characteristics within a program that will be helpful in the identification of a corresponding programmer, and whose computation can be automated with a reasonable cost.

What makes us believe that identification of authorship in computer software is possible? People work within certain repeated frameworks. They use those things that they are more comfortable with or are accustomed to. Humans are creatures of habit, and habits tend to persist. That is why, for example, we have a handwriting style that is consistent during periods of our life, although the style may vary as we grow older.

Likewise for programming, we can ask: which are the programming constructs that a programmer uses all the time? These are the habits that will be more likely entrenched, the things he consistently and constantly does and that are likely to become ingrained.

3 Motivation.

Spafford and Weeber suggested that it might be feasible to analyze the remnants of software, typically the remains of a virus or trojan horse, and identify its author. They theorized that this technique, called Software Forensics, could be used to examine and analyze software in any form; be it source code for any language or executable images [WS93].

Among the measurements that Spafford and Weeber suggest are the preference for certain data structures and algorithms, the compiler used, the level of expertise of the author of a program, the choice of system calls made by the programmer, the formatting of the code, the use of pragmas or macros that might not be available on every system, the commenting style used by the programmer, variable naming convention used, and misspelling of words inside comments and variables. However, Spafford and Weeber provided no statistical evidence that might support their theory.

The work presented on this paper is precisely an attempt at determining the validity of their assumptions. If proven right, four basic areas can benefit from the development of authorship analysis techniques: 1) For authorship disputes, the legal community is in need of methodologies that can be used to provide empirical evidence to show that two or more programs are written by the same person. 2) In the academic community, it is considered unethical to copy programming assignments. While plagiarism detection can show that two programs are equivalent, authorship analysis can be used to show that some code fragment was indeed written by the person who claims authorship of it. 3) In industry, where there are large software products that typically run for years, and contain millions of lines of code, it is a common occurrence that authorship information about programs or program fragments is nonexistent, inaccurate or misleading. Whenever a particular program module or program needs to be rewritten, the author may need to be located. It would be convenient to be able to determine the name of the programmer who wrote a particular piece of code from a set of several hundred programmers so he can be located to assist in the upgrade process. 4) Real-time intrusion detection systems could be enhanced to include authorship information[2] [Krs94]

4 Survey of Related Work

In literature, the question of Shakespeare's identity has engaged the wits and energy of a wide range of people for more than two hundred years. Such great figures as Mark Twain, Irving Wall and Sigmund Freud debated at length this particular issue [HH92].

Hundreds of books and essays have been written on this topic, some as early as 1837 [Dis37]. Especially interesting was W. Elliott's attempt to resolve the authorship of Shakespeare's work with a computer by examining literary minutiae from word frequency to punctuation and proclivity to use clauses and compounds[EV91]. Although much controversy surrounds the specific results obtained by Elliott's computer analysis, it is clear from the results that works attributed to Shakespeare fit a narrow and distinctive profile[Krs94].

The issue of identifying program authorship was ex-

[2] A programmer signature constructed from the identifying characteristics of programs constitutes a pattern that can be used in the monitoring of abnormal system ussage.

plored by Cook and Oman [OC89] as a means for determining instances of software theft and plagiarism[3]. They briefly explored the use of software complexity metrics to define a relationship between programs and programmers, concluding that these are inadequate measures of stylistic factors and domain attributes. Two other studies by Berghel and Sallach [BS84] and Evangelist [Eva84] support this theory.

Cook and Oman explain the use of "markers" to describe the occurrences of certain peculiar characteristics, much like the markers used to resolve authorship disputes of written works. The markers used in their work are based purely on typographic characteristics.

For collecting data to support their claim, they built a Pascal source code analyzer that generated an array of Boolean measurements based on commenting style, indentation, lower case characters only, upper case characters only, multiple statements per line, blank lines in program body, whether case was used to distinguish between keywords and identifiers, whether underscores were used in identifiers, whether the BEGIN keyword was followed by a statement on the same line, and whether the THEN keyword was followed by a statement on the same line

To test their hypothesis, Cook and Oman collected the metrics mentioned above for eighteen short programs by six authors. The programs were taken from example code for three tree-traversal algorithms and one simple sorting algorithm.

Cook and Oman claim that the results of the experiment were surprisingly accurate. The results are encouraging, but further reflection shows that the experiment is fundamentally flawed. It fails to consider that textbook algorithms are frequently cleaned by code beautifiers and pretty printers, and that different problem domains will demand different programming methodologies. The implementation of the three tree-traversal algorithms involves only slight modifications and hence are likely to be similar.

Spafford and Weeber suggested that it might be feasible to analyze the remnants of software, typically the remains of a virus or trojan horse, and identify its author. This technique, called Software Forensics, could be used to examine and analyze software in any form; be it source code for any language or executable images [WS93]. Software Forensics is really a superset of authorship analysis using style analysis because some of the measurements suggested by Spafford and Weeber include, but are not limited to, some of the measurements made by Cook and Oman. The list of measurements suggested by Spafford and Weeber is comprehensive, but the derivation of some of these are difficult to automate[Krs94].

[3]It is important to realize that authorship analysis is markedly different from Plagiarism Detection. Plagiarism detection can not tell if two entirely different programs were written by the same person. Also, the replication need not maintain the programming style of the original software. Many people have devoted time and resources to the development of plagiarism detection [Ott77, Gri81, Jan88, Wha86], and a comprehensive analysis of their work is beyond the scope of this paper.

5 Difficulties in Authorship Analysis

We expect the programming characteristics of programmers to change and evolve. Education is only one of many factors that have an effect on the evolution of programming styles. Not only do software engineering models impose particular naming conventions, parameter passing methods and commenting styles; they also impose a planning and development strategy. The waterfall model [GJM91], for example, encourages the design of precise specifications, utilization of program modules and extensive module testing. These have a marked impact on programming style.

The programming style of any given programmer varies also from language to language, or because of external constraints placed by managers or tools[4]. Out of the set of measurements that allow our model to identify the authorship of a program, can we identify those that have been contaminated and ignore them for our analysis? A good example would be the analysis of code that has been formatted using a pretty-printer. Would it be possible for the authorship analysis system to recognize that such a formatter has been used, identify the pretty-printer and compensate by eliminating information about indentation, curly bracket placement and comment placement? Conceptually similar would be the recognition of tools used that force onto the programmer a particular programming style. For example, could the authorship analysis tool recognize the usage of Motif and compensate for variable naming conventions imposed by the tool?

Finally, among the most serious problems that must be resolved with authorship analysis is the reuse of code. All the work performed up to date on this subject assumes that a significant part of the code being analyzed was built and developed by a single individual. In commercial development projects, this is seldomly the case.

[4]The use of the Motif, GL, PLOT-10 or GKS libraries generally demands that the application be structured in some fashion or may impose naming conventions.

6 Experimental Setup

The term "Software Metric" was defined by Conte, Dunsmore and Shen in [SCS86] as: "*Software metrics are used to characterize the essential features for software quantitatively, so that classification, comparison, and mathematical analysis can be applied.*" What we are trying to measure, for establishing the authorship of a program, is precisely some of these features. Hence, the term software metric, or simply metric, is appropriate to describe these special characteristics.

Although theoretically possible, it would be impractical to compare style metrics across different development platforms. Among similar languages like C, Modula and Pascal, the same metrics might be used successfully with similar results. This might not be true if C is compared with Prolog or LISP. These programs belong to three different programming paradigms (Structured Programming, Logic Programming and Functional Programming) and there are large differences among them. Many of the metrics we could use for identifying authorship in one of these programming languages will be of no use in the others.

Hence, in this paper we will limit ourselves to the stylistic concerns of C source code. Programmers are comfortable using it and the language is commonly used in the academic community and in industry.

7 Sources for the Collection of Metrics

We can collect metrics for authorship detection from a wide variety of sources:

- Oman and Cook [OC91] collected a list of 236 style rules that can be used as a base for extracting metrics dealing with programming style.
- Conte, Dunsmore and Shen [SCS86] give a comprehensive list of software complexity metrics.
- Kernighan and Plauger [KP78] give over seventy programming rules that should be part of "good" programming practice.
- Van Tassel [Tas78] devotes a chapter to programming style for improving the readability of programs.
- Jay Ranade and Alan Nash [RN93] give more than three hundred pages of style rules specifically for the "C" programming language.
- Henry Ledgard gives a comprehensive list of "C" programming proverbs that contribute to programming excellence [Led87].

Many other sources have influenced our choice of metrics [LC90, BB89, OC90b, Coo87] but do not contain a specific set of rules, metrics or proverbs.

All these sources give us ample material to select the metrics we will use. Because of the large number of rules and metrics available, we have decided to divide our metrics into three categories.

We would like to examine those metrics that deal specifically with the layout of the program. In this category we will include such metrics as the ones that measure indentation, placement of comments, placement of brackets, etc. We will call these metrics "Programming Layout" metrics.

All these metrics are fragile because the information required can be easily changed using code formatters and pretty printers. Also, the choice of editor can significantly change some of the metrics of this type. Emacs, for example, encourages consistent indentation and curly bracket placement. Furthermore, many programmers learn their first programming language in university courses that impose a rigid and specific set of style rules regarding indentations, placement of comments and the like [MB93].

Also useful are the metrics that deal with characteristics that are difficult to change automatically by pretty printers and code formatters, and are also related to the layout of the code. In this category we include those metrics that measure mean variable length, mean comment length, etc. We will call these metrics "Programming Style" metrics.

Finally, we would like to examine metrics that we hypothesize are dependent on the programming experience and ability of the programmer. In this category we will find such metrics as mean lines of code per function, usage of data structures, etc. We will call these metrics "Programming Structure" metrics,

8 Metrics Considered

>From all the sources mentioned in Section 7, we extracted a series of potentially useful software metrics. Even though we describe these metrics as indivisible measurements, in practice we might calculate several values for them. For example, for metric STY1a we might calculate a median and a standard error. We will examine the exact format in greater detail in later sections.

Also, unless explicitly stated, all the metrics consider only the text inside function bodies. We don not examine include files or type declarations because there is no way

of differentiating between those declarations that are imported from external modules, and those that are native to the programmer.

8.1 Programming Layout Metrics

- Metric STY1: A vector of metrics indicating indentation style [RN93, pages 68–69]:
  - Metric STY1a: Indentation of C statements within surrounding blocks.
  - Metric STY1b: Percentage of open curly brackets ({) that are alone in a line.
  - Metric STY1c: Percentage of open curly brackets ({) that are the first character in a line.
  - Metric STY1d: Percentage of open curly brackets ({) that are the last character in a line.
  - Metric STY1e: Percentage of close curly brackets (}) that are alone in a line.
  - Metric STY1f: Percentage of close curly brackets (}) that are the first character on a line.
  - Metric STY1g: Percentage of close curly brackets (}) that are the last character in a line.
  - Metric STY1h: Indentation of open curly brackets ({).
  - Metric STY1i: Indentation of close curly brackets ({).
- Metric STY2: Indentation of statements starting with the "else" keyword.
- Metric STY3: In variable declarations, are variable names indented to a fixed column?
- Metric STY4: What is the separator between the function names and the parameter lists in function declarations? Possible values are spaces, carriage returns or none.
- Metric STY5: What is the separator between the function return type and the function name in function declarations? Possible values are spaces or carriage returns.
- Metric STY6: A vector of metrics that will help identify the commenting style used by the programmer. The vector will be composed of:
  - Metric STY6a: Use of borders to highlight comments.
  - Metric STY6b: Percentage of lines of code with inline comments.
  - Metric STY6c: Ratio of lines of block style comments to lines of code.
- Metric STY7: Ratio of white lines to lines of code [RN93, pages 70–71].

8.2 Programming Style Metrics

- Metric PRO1: Mean program line length (characters per line) [BM85].
- Metric PRO2: A vector of metrics that will consider name lengths.
  - Metric PRO2a: Mean local variable name length.
  - Metric PRO2b: Mean global variable name length.
  - Metric PRO2c: Mean function name length.
  - Metric PRO2d: Mean function parameter length.
- Metric PRO3: A vector of metrics that will tell us about the naming conventions chosen by the programmer. This vector will consist of:
  - Metric PRO3a: Some names use the underscore character.
  - Metric PRO3b: Use of temporary variables[5] that are named XXX1, XXX2, etc. [KP78], or "tmp," "temp," "tmpXXX" or "tempXXX" [RN93].
  - Metric PRO3c: Percentage of variable names that start with an uppercase letter.
  - Metric PRO3d: Percentage of function names that start with an uppercase letter.
- Metric PRO4: Global variable count to mean local variable count ratio. This metric could potentially tell us something about the programmer's propensity to use global variables.
- Metric PRO5: Global variable count to lines of code ratio. This variation of the previous metric might give us a better metric for measuring the frequency of usage of global variables.
- Metric PRO6: Use of conditional compilation.
- Metric PRO7: Preference of either "while," "for" or "do" loops.
- Metric PRO8: Does the programmer use comments that are nearly an echo of the code [KP78, page 143] [RN93, page 82]?

[5] It can be argued that all local variables are temporary and no global variable is temporary. However, in this paper we will follow the convention that a variable is temporary if it there is no direct association between its name and its meaning.

- Metric PRO9: Type of function parameter declaration. Does the user use the standard format or the ANSI C format?

8.3 Programming Structure Metrics

- Metric PSM1: Percentage of "int" function definitions.
- Metric PSM2: Percentage of "void" function definitions.
- Metric PSM3: Program uses a debugging symbol or keyword[6]. We would specifically be looking at identifiers or constants containing the words "debug" or "dbg" [RN93, pages38–53].
- Metric PSM4: The assert macro is used.
- Metric PSM5: Lines of code per function [KP78, BM85].
- Metric PSM6: Variable count to lines of code ratio. This metric could identify those programmers who tend to avoid reusing variables, creating new variables for each loop control variable, etc.
- Metric PSM7: Percentage of global variables that are declared static.
- Metric PSM8: The ratio of decision count to lines of code. To simplify the computation of this metric, we have chosen to modify the definition of decision count as given in [SCS86]. We do not count each logical operator inside a test as a separate decision. Rather, each instance of the if, for, while, do, case statements and the ? operator increases our decision count by one.
- Metric PSM9: Is the goto keyword used? Software designers and programmers still rely on these [BM85].
- Metric PSM10: Simple software complexity metrics offer little information that might be application independent [OC89]. The metrics that we could consider are: cyclomatic complexity number, program volume, complexity of data structures used, mean live variables per statement, and mean variable span [SCS86].
- Metric PSM11: Error detection after system calls that rarely fail. Some programmers tend to ignore the error return values of system calls that are considered reliable [GS92, page 164]. Thus, a metric can be obtained out of the percentage of reliable system calls whose error codes are ignored by the programmer. Also, some programmers tend to overlook the error codes returned by system calls that should never have them ignored (like "malloc"). We can define this metric as a vector of the following items:
  - Metric PSM11a: Are error results from memory related system calls ignored? Specifically, we would be looking at malloc(), calloc(), realloc(), memalign(), valloc(), alloca() and free().
  - Metric PSM11b: Are error results from I/O routines ignored? Specifically, we would be looking at open(), close(), dup(), lseek(), read(), write(), fopen(), fclose(), fwrite(), fread(), fseek(), getc(), putc(), gets(), puts(), printf() and scanf().
  - Metric PSM11c: Are error results from other system calls ignored? We would be looking at chdir(), mkdir(), unlink(), socket(), etc.
- Metric PSM12: Does the programmer rely on the internal representation of data objects? This metric would check for programmers relying on the size and byte order of integers, the size of floats, etc.
- Metric PSM13: Do functions do "nothing" successfully? Kernighan and Plauger in [KP78, pages 111–114] and Jay Ranade and Alan Nash in [RN93, page 32] emphasize the need to make sure that there are no unexpected side effects in functions when these must "do nothing." In this context, functions that "do nothing" successfully are functions that correctly test for boundary conditions on their input parameters. We must note that it is an undecidable problem to determine the correctness of an arbitrary function[HU79].
- Metric PSM14: Do comments and code agree? Kernighan and Plauger write in [KP78] that "A comment is of zero (or negative) value if it is wrong". Ranade and Nash [RN93, page 89] devote a rule to the truth of every comment. Even if the comments were initially accurate, it is possible that during the maintenance cycle of a program they became inaccurate. Because we cannot determine the stage of development where the incorrect comment was introduced, we will consider all incorrect comments[7] in this metric.
- Metric PSM15: More than any other type of software metric, those that deal with the development phase of

[6] Debugging is difficult. Many non standard techniques have been developed [RN93], and we cannot hope to identify all forms of debugging symbols. However, there are some techniques that are widely used and we will concentrate on these.

[7] Deciding that a comment is wrong can only be done manually by careful examination of the source code. Because it involves the semantic analysis of English sentences, it is unlikely that this process will be automated soon.

a project would help to identify the authorship of a program. Consider, for example, whether comments are placed before, during or after the development of a program, the choice of editor, the choice of compiler, the usage of revision control systems, the usage of development tools, etc. Unfortunately, this information is not readily available. Test programs, intermediate versions, debugging code and the alike are discarded after the final version of the program is finished.

- Metric PSM16: Quality of software. We could use software metrics that deal with the quality of software to assess the level of experience of the programmer. Typically, software quality metrics are related to software development standards and try to measure the reliability and robustness of software.
  These metrics will not be useful. In the worst case, we would be measuring the care that the programmer has taken to develop a piece of code as well as the level of expertise of the programmer. Furthermore, it is possible for an experienced programmer to get low software quality scores and for a beginner to get high scores (if he followed a textbook algorithm for his program).

A software analyzer built for the lcc C compiler front-end developed at Princeton [Han91] was used to generate most of the raw data, including all of the programming structure metrics. Once the calculation of these metrics had been performed, a series of Perl programs were used to collect the metrics that depended on the information that was discarded by the C preprocessor. Indentation, commenting style and line lengths are examples of the measurements collected by these scripts.

9 Experimental Stages

The experimental data for this paper was gathered in three distinct stages: a preliminary stage helped us determine the proper methods for calculating the metrics, eliminated those metrics that were clearly inappropriate for our purposes, and coexisted with the tool development phase[Krs94]; a pilot experiment was performed with a small number of programmers, each of whom wrote three short and simple programs [Krs94]. To determine the effect of problem domains on our analysis, the programs were oriented to the three areas where we thought we could have the greater variations in style: computationally intensive programs, I/O intensive programs and data structure intensive programs[8]; once the preliminary experiment showed that the desired set of metrics could be analyzed, we designed and executed a larger, more formal experiment in which to test our prototype.

9.1 Experiment

For this experiment, a series of programs were collected from a total of 29 students, staff and faculty members at Purdue University. The distribution for the programs are shown in table 1.

Table 1: Distribution of Programs

Group Identification	Programs
Students 1(Projects for the Fall 1993 term)	57
Students 2 (Programs developed for other terms)	6
Pilot 1 (Programs developed by students for the pilot experiment)	18
Pilot 2 (Programs developed by experienced programmers for the pilot experiment)	6
Faculty (Miscellaneous programs by faculty members)	7
TOTAL	88

We included programs from a wide variety of programming styles and for different problem domains. Roughly one third of the student programs were programming assignments from a graduate level networking course, one third of the programs were programming assignments from a graduate level compilers course and one third of the programs were from miscellaneous graduate level courses, including data bases, numerical analysis and operating systems. Of the programs submitted by the faculty members, half are oriented towards numerical analysis and half oriented towards compiler construction and software engineering.

9.2 Statistical Model Used for the Analysis

There are two statistical methods that could be used to analyze the metrics gathered. Cluster analysis, as used by Oman and Cook in [OC89] can only be used if we

[8] A detailed description of the programs can be found in [Krs94]

discretize the values for our metrics. Unfortunately, it is difficult to find ranges for each of the metrics that could be used for any group of programmers without loss of accuracy.

The second statistical analysis method we can use, and the one chosen for our analysis, is discriminant analysis. This method, described in [SAS, JW88] is a multivariate technique concerned with separating observations and with allocating new observations into previously defined groups.

Originally, for the final experiment we wanted to keep those metrics that showed little variation between programs (for a specific programmer) and those metrics that showed large variations among programmers. Unfortunately, analysis of the metrics collected shows that these two criteria are not necessarily correlated.

Initially, we calculated the standard error by programmer for every metric, and eliminated those that showed large variations because they identify those style characteristics where the programmer is inconsistent.

Surprisingly, most of the metrics that showed large variations among programmers were eliminated as well. The performance of our statistical analysis with the remaining metrics was discouraging, with only twenty percent of the programs being classified correctly.

The step discrimination tool provided by the SAS program [SAS] should theoretically be capable of eliminating bad metrics from the statistical base. Unfortunately, this tool was not helpful because it failed to eliminate any of the metrics from our set.

To resolve this issue, we decided to build a tool that would help us visualize the metrics collected. For each continuous metric (i.e. real valued metric) the tool displayed two graphs that showed the variation of the metric within programs for each programmer and the distribution of values for each metric for all programmers.

For each discrete metric (i.e. boolean metrics and set metrics), the tool produced a graph that showed the consistency of each programmer for each metric. In these figures, vertical lines represent a programmer "jumping" from one value to the next in two consecutive programs. Hence, a good discrete metric is one that shows variations in values and no "jumps."

With this analysis, we chose a small subset of our metrics for the final statistical analysis[9].

[9]Specifically, metrics PRO1M, mean for PRO2a, mean for PRO2b, mean for PRO2c, PRO3d, PRO5, PSM1, PSM6, mean for STY1a, STY1b, STY1c, STY1d, STY1e, STY1f, mean for STY1i, mean for STY2, STY6b, STY6c, STY7, PRO8, PSM3, STY4 and STY5.

9.3 Experiment Results

The success rate of our experiment is 73%. This means that of all the programs analyzed, 73% were correctly assigned to their original programmers. Individual percentages of correctly classified programs are shown in table 2.

Table 2: Classification by Programmer

% of programs	Number of
% correctly classified	programmers
100 %	17
77 %	3
75 %	1
71 %	1
50 %	1
33 %	2
25 %	1
20 %	1
0 %	2

When colleagues were shown this table for the first time, the first question asked was: "Are all the programmers that the system identified correctly 100% of the time related? Are the backgrounds of these programmers similar?" Initially we were surprised to see that the programs for seasoned programmers, a faculty member, and graduate students of Computer Science were all mixed in this category. Also, we notice that:

1. The programs for the faculty member (three programs averaging 300 lines of code each) were developed over several years and address different problem domains.
2. Three of the six programmers who helped with the development of the programs for the pilot study were correctly classified 100% of the time. The programs for each of these programmers addressed different problem domains.
3. The programmers who were correctly classified have different backgrounds.

For the programmers who were classified less than 50% of the time, we looked at their code to find out why we failed to classify them (two programmers were never classified correctly). We were surprised to find that they had varied their programming style considerably from program to program in a period of only two months.

Other misclassified programmers showed a consistent programming style. This fact is a clear indication that the metrics chosen for our experiment were not comprehensive enough to distinguish among them. But their programs are far from identical as subsequent inspection of their code revealed. For one of the programmers who was classified correctly 0% of the time, for example, we could find several characteristics that remained consistent throughout.

Our experiment also helped us predict the performance of the metrics when a program not included in the original database is considered. For each program, we removed it from the database and later told SAS to classify it. As expected, the results average 73 %. However, this stage of our experiment shed some light as to the consistency of the misclassification. Mainly, some programmers are misclassified consistently. Programer 18 was misclassified consistently as programmer 12, programmer 19 as programmer 17, programmer 11 as programmer 18, and programmer 26 as programmer 9. We can conclude that even though the metrics are not good enough to classify these programmers correctly, the misclassification is not random. A more refined set of metrics could help distinguish among these programmers.

The statistical analysis tools used provide little support for ranking the performance of individual metrics. The removal of any one metric from the analysis can have negative or positive effects, independent of the quality of the metric.

10 Conclusions

The experiments we have performed for this paper support the theory that it is possible to find a set of metrics that can be used to classify programmers correctly. Close visual examination of the source code provided by all the programmers involved in our experiment reveals that programmers tend to show repeating patterns in their programs.

Clearly it is possible to identify ownership of a program by examining some finite set of metrics. As expected, programmers are skillful with a limited set of constructs, mainly those that are well known to them and that allow them to write programs faster and more reliably. It would be unrealistic to assume that any programmer can develop programs efficiently and correctly using an unfamiliar programming style. This does not only apply to the structure of the programs, but also to the look and feel of it; such metrics as, for example, average blank lines over lines of code can indeed remain surprisingly constant. Programmers organize information on the screen such that logically independent portions of the code can be easily recognized.

Even though we are satisfied with our choice of metrics, the results presented in this paper clearly show that we will not be able to correctly classify all possible programmers successfully with this set of metrics. Experience and logic tell us that a small and fixed set of metrics are not sufficient to detect ownership of every program and for every programmer.

Our results suggest that using weighted combinations of metrics might produce better results. We also suspect that using Bayesian measures for some combinations of metrics might also produce better results. This would require a more involved study to determine appropriate prior probabilities and dependencies.

By no means do we claim that the set of metrics we examined is the only one that might yield stable measurements. During the data collection and analysis of the experiment, we noted that the following metrics might be of considerable use in future experiments:

1. Use of revision control system headers. We were surprised to see that a considerable portion of the programmers examined used the automatic identification and log features of the RCS Revision Control System. As an added bonus, such identification strings will provide the login name of the programmer in question[10].
2. Another metric that could have been used successfully is the use of literals in code versus the use of global constants.
3. One programmer's idea of debugging statements was commenting out the print statements. This was done consistently and it might provide another useful metric.

We do not expect that the metrics calculated for any given programmer would remain an accurate tag for a programmer for a long time, even though in our experiment we have correctly identified the only programmer who provided code developed over a number of years. Further research must be performed to examine the effect that time and experience has on the metrics examined on this document.

It would be logical to conclude that for the authorship

[10] It is easy to alter the user name in the RCS automatic identification feature, and as such, excessive confidence must not be placed on its accuracy

analysis techniques to work, the metrics would have to be gathered continually over time. Compilers and operating systems would have to be enhanced and significant research would have to be done in the development of operating systems to enforce the use of these metrics.

The results of this paper support the conclusion that within a closed environment, and for a specific set of programmers, it is possible to identify a particular programmer and the probability of finding two programmers that share exactly those same characteristics should be small.

Acknowledgements

This work was partially supported by a gift from Sun Microsystems to the COAST Laboratory: that support is gratefully acknowledged.

References

[BB89] A. Benander and B. Benander. An empirical study of COBOL programs via a style analyzer: The benefits of good programming style. *The Journal of Systems and Software*, 10(2):271–279, 1989.

[BM85] R. Berry and B. Meekings. A style analysis of C programs. *Communications of the ACM*, 28(1):80–88, 1985.

[BS84] H. Berghel and D. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 19(8):65–76, 1984.

[Coo87] Doug Cooper. *Condensed Pascal*. W. W. Norton and Company, 1987.

[Dau90] K. Dauber. *The Idea of Authorship in America*. The University of Wisconsin Press, 1990.

[Den87] D. Denning. An intrusion detection system. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[Dij68] E. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[Dis37] B. Disraeli. *Venetia*. New York and London, 1837.

[EV91] W. Elliot and R. Valenza. Was the Earl of Oxford the true Shakespeare? *Notes and Queries*, 38:501–506, December 1991.

[Eva84] M. Evangelist. Program complexity and programming style. In *Proceedings of the International Conference of Data Engineering*, pages 534–541. IEEE, 1984.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, first edition, 1991.

[Gri81] S. Grier. A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bulletin*, 13(1):15–20, 1981.

[GS92] S. Garfinkel and E. Spafford. *Practical Unix Security*. O'Reilly & Associates, Inc., 1992.

[Han91] D. Hanson. Code generation interface for ANSI C. *Software - Practice and Experience*, 38:963–988, September 1991.

[HH92] W. Hope and K. Holston. *The Shakespeare Controversy*. McFarland & Company, 1992.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, first edition, 1979.

[Jan88] H. Jankowitz. Detecting plagiarism in student Pascal programs. *Computer Journal*, 31(1):1–8, 1988.

[JW88] R. Johnson and D. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, second edition, 1988.

[KP78] B. Kernighan and P. Plauger. *The Elements of Programming Style*. McGraw-Hill Book Company, second edition, 1978.

[KR85] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1985.

[Krs94] Ivan Krsul. Authorhip analysis: Identifying the author of a program. Master's thesis, Department of Computer Science, Purdue University, 1994.

[LC90] A. Lake and C. Cook. STYLE: An automated program style analyzer for Pascal. *ACM SIGCSE Bulletin*, 22(3):29–33, 1990.

[Led87] Henry Ledgard. *C With Excellence: Programming Proverbs*. Hayden Books, 1987.

[MB93] R. Madison and M. Beaven. *FORTRAN For Scientists and Engineers: Laboratory Manual*. McGraw-Hill, Inc., 1993.

[NS86] T. Naps and B. Singh. *Introduction to Data Structures with Pascal*. West Publishing Company, 1986.

[OC89] P. Oman and C. Cook. Programming style authorship analysis. In *Seventeenth Anual ACM Computer Science Conference Proceedings*, pages 320–326. ACM, 1989.

[OC90a] P. Oman and C. Cook. A taxonomy for programming style. In *Eighteenth Anual ACM Computer Science Conference Proceedings*, pages 244–247. ACM, 1990.

[OC90b] P. Oman and C. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, 1990.

[OC91] P. Oman and C. Cook. A programming style taxonomy. *Journal of Systems Software*, 15(4):287–301, 1991.

[Ott77] K. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 1977.

[RN93] J. Ranade and A. Nash. *The Elements of C Programming Style*. McGraw-Hill Inc., 1993.

[SAS] The SAS Institute. *SAS/STAT User's Guide. Volume 1, ANOVA–FREQ*, fourth edition.

[SCS86] H. Dunsmore S. Conte and V. Shen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, 1986.

[Set89] R. Sethi. *Programming Languages Concepts and Constructs*. Addison–Wesley Publishing Company, 1989.

[Spa89] E. Spafford. The internet worm program. Technical Report CSD-TR-823, Department of Computer Science. Purdue University, 1989.

[Sto90] C. Stoll. *The Cuckoo's Egg*. Pocket Books, first edition, 1990.

[Tas78] Dennie Van Tassel. *Program Style, Design, Efficiency, Debugging, and Testing*. Prentice Hall, 1978.

[Wha86] G. Whale. Plague: Detection of plagiarism using program structure. In *Proceedings of the Ninth Australian Computer Science Conference*, pages 231–241, 1986.

[WS93] Stephen A. Weeber and Eugene H. Spafford. Software forensics: Can we track code to its authors? *Computers & Security*, 12(6):585–595, December 1993.