Authorship Analysis: Identifying The Author of a Program[1]

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

May 3, 1994
Technical Report CSD–TR–94–030

[1]This paper was originally written as a Master's thesis at Purdue University.

Abstract

In this paper we show that it is possible to identify the author of a piece of software by looking at stylistic characteristics of C source code. We also show that there exist a set of characteristics within a program that are helpful in the identification of a programmer, and whose computation can be automated with a reasonable cost.

There are four areas that benefit directly from the findings we present herein: the legal community can count on empirical evidence to support authorship claims, the academic community can count on evidence that supports authorship claims of students, industry can count on identifying the author of previously un-identifiable software modules, and real time intrusion detection systems can be enhanced to include information regarding the authorship of all locally compiled programs.

We show that it is possible to identify the author of a piece of software by collecting and identifying eighty-eight programs for twenty nine students, staff and faculty members at Purdue University.

Chapter 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 the software used for the attack, we might want to identify the source of the software. Typical examples of such software are Trojan horses[1], viruses[2], and logic bombs[3].

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.

This seems at first an impossible task, and convincing arguments can be given about the intractability of this problem. Consider, for example, the following short list of potential problems with the identification of authors:

1. Given that millions of people write software, it seems unlikely that, given a piece of software, we will find the programmer who wrote it.

2. Software evolves over time. As time passes, programmers vary their programming habits and their choice of programming languages. The

[1]Trojan horses are defined in [GS92] as programs that appear to have one function but actually perform another function.

[2]Viruses are defined in [GS92] as programs that modify other programs in a computer, inserting copies of themselves.

[3]Logic bombs are defined in [GS92] as hidden features in programs that go off after certain conditions are met.

development of new software engineering methods, the introduction of formal methods for program verification, and the development of user-friendly, graphic oriented code processing systems and debuggers all contribute to making programming a highly dynamic field.

3. Software gets reused. In recent years, and with the development of object oriented programming methodologies, programmers have come to depend on reusing large portions of code; similar to the code produced by the GNU/Free Software Foundation[4], much of it is public domain.

   Commercially available prototypers, like the Builder Xcessory by Integrated Computer Solutions, Inc., produce thousands of lines of code that are used to develop Motif interfaces. Similar development tools are available for hundreds of development platforms.

Similar arguments could be given for fingerprinting: Fingerprint matching is an expensive process and it seems unlikely that government agencies will ever be able to classify every citizen in their lifetime. It is also unlikely that given a fingerprint, we will be able to pick from a pool of several million people the correct person every time.

The identification process in computer software can be made reliable for a subset of the programmers and programs. Programmers that are involved in high security projects or programmers that have been known to break the law are attractive candidates for classification.

1.1 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 have 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.

[4] The Free Software Foundation is a group started by Richard Stallman to embody his ideas of personal freedom and how software should be produced.

Our objective is to classify the programmer and to try to find a set of characteristics that remain constant for a 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 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 frameworks that repeat themselves. They use those things that they are more comfortable with or are accustomed to.

Programmers are humans. 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. Patterns of behavior are all around us.

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.

1.2 Motivation.

Four basic areas can benefit considerably by the development of solid authorship analysis tools:

1. For authorship disputes, the legal community is in need of solid 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 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. Dorothy Denning writes in [Den87] about a proposed real-time intrusion detection system:

   > The model is based on the hypothesis that exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations could be detected from abnormal patterns of system usage.

   Obviously, a programmer signature constructed from the identifying characteristics of programs constitutes such a pattern. For example, consider the student who retrieves a copy of a password cracking program and compiles it in his university account. Once the compiler collects the identifying features of the program, the operating system could immediately recognize the compiling of this program as an abnormal event.

   Of course we realize that it is theoretically possible for a programmer to fool or bypass the system by altering his programming methods. The change would have to be gradual, subtle and ingenious for the system not to record the change. However, we expect that modifying a user's profile to meet the characteristics of a specific program (a password cracking program, for example) will be a difficult and time consuming process.

Researchers who seriously consider this issue must also address the redesign of compilers, interpreters and operating systems to enforce the use of these metrics at run time. A compiler is just another program. What is to prevent someone from bringing or building his own compiler piece by piece?

Operating systems must also prevent the installation of programs compiled on other systems by potential penetrators. To maintain consistency, all executable programs in the system must be either compiled locally, or registered with the system administrator. If this step was not enforced, any programmer might compile source code on a privately owned machine, importing executable programs with false identification information.

We must also consider that the data that authorized compilers generate must be protected. If such identifying information is stored in the binary file, what prevents someone from writing a program to change it?

1.3 Desired Results

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 ownership of any program.

The implications of being able to find such a signature are enormous. Cliff Stoll's German hacker [Sto90] never feared prosecution precisely because of our inability to identify him even after tracking him down (the hacker had to be caught red-handed to be prosecuted). The author of the Internet Worm that attacked several hundred machines in the evening of November 2, 1988 [Spa89] might also have been identified quickly since he was a student and many samples of his programming style were readily available. Authors of computer viruses might also be identified if a piece of the source code is available.

1.4 Summary.

Identifying the author of a program is a difficult process. We need some powerful tools to discover the identity of programmers who are not explicitly named or who might simply wish to conceal themselves. These tools must prove reliable.

In the remainder of this paper, we will put forth a methodology for identifying such source code. Chapter 2 introduces much of the background material and explores the work that has been done in related areas. Chapter 3 introduces the methodology and experimental setup. Chapter 4 details the experiments performed and outlines the findings. Finally, Chapter 5 presents the conclusions and details the work that must be done in the future.

Chapter 2

Authorship Analysis

2.1 Definitions

An *Author* is defined by Webster [MW92] as "one that writes or composes a literary work," or as "one who originates or creates." In the context of software development we adapt the definition of author to be: "one that originates or creates a piece of software." *Authorship* is then defined as, "the state of being an author." As in literature, a particular work can have multiple authors. Furthermore, some of these authors can take an existing work and add things to it, evolving the original creation.

A *program* is a specification of a computation [Set89] or alternatively, a sequence of instructions that permit a computer to perform a particular task [Spe83]. A *programming language* is a notation for writing programs [RR83, Set89].

A *programmer* is a person who designs, writes and tests computer programs [Spe83]. In the fullest meaning of the term, a programmer will participate in the definition and specification of the problem itself, as well as the algorithms to be used in its solution. He will then design the more detailed structure of the implementation, select a suitable programming language and related data structures and write and debug the necessary programs [RR83].

Programming style is a distinctive or characteristic manner present in those programs written by a programmer.

A predicate is a propositional formula in which relational and quantified expressions can be used in place of prepositional variables. To interpret a predicate, we first interpret each relational and quantified expression—yielding true and false for each—and then interpret the resulting prepositional formula [And91, page 20].

A complex system may be divided into simpler pieces called *modules*.

While writing software that is divided into modules, we can: look at the details of each module in isolation, ignoring other modules until the module has been tested and completed, or look at the general characteristics of each module and their relationships to integrate them. If these two phases are executed in this order, then we say that the system is designed *bottom up*; the converse denotes *top-down* design.

Prototypers are programs that allow users to create automatically large portions of code, generally for the design of user-interfaces, relieving the user of the monotonous production of well understood and largely standard code.

2.2 Survey of Related Work

2.2.1 Authorship Analysis in Literature

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]. Mark Twain, for example, made the following observations about Shakespeare and his writings:

- Shakespeare's parents could not read, write or sign names.
- He made a will, signed on each of the three pages. He went to great lengths to distribute his wealth among the members of his family. This will was a businessman's will and not a poet's.
- His will does not mention a single book, nor a poem nor any scrap of manuscript of any kind.
- There is no other known specimen of his penmanship that can be proved his, except one poem that he wrote to be engraved on his tomb [Nei63].

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.

For three years, the Shakespeare Clinic of Claremont Colleges used computers to see which of fifty-eight claimed authors of Shakespeare's works matched Shakespeare's writing style [EV91]. Among the techniques used was a modal test that divided a text into blocks, fifty-two keywords in each block, and measured and ranked eigenvalues, or modes. Rather than representing keyword occurrences, modes measure patterns of deviation from a writer's rates of word frequency. Other more conventional tests examined were hyphenated compound words; relative clauses per thousand; grade-level of writing; and percentage of open – and feminine – ended lines [EV91].

Although much controversy surrounds the specific results obtained by Elliott's computer analysis, it is clear from the results that Shakespeare fits a narrow and distinctive profile. As W. Elliot and R. Valenza write in [EV91]:

> Our conclusion from the clinic was that Shakespeare fit within a fairly narrow, distinctive profile under our best tests. If his poems were written by a committee, it was a remarkably consistent committee. If they were written by any of the claimants we tested, it was a remarkably inconsistent claimant. If they were written by the Earl of Oxford, he must, after taking the name of Shakespeare, have undergone several stylistic changes of a type and magnitude unknown in Shakespeare's accepted works.

2.2.2 Authorship Analysis With Programming Style

The issue of identifying program authorship was explored by Cook and Oman in [OC89] as a means for determining instances of software theft and plagiarism. 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 describe 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:

1. Inline comments on the same line as source code.
2. Blocked comments (two or more comments occurring together).
3. Bordered comments (set off by repetitive characters).
4. Keywords followed by comments.
5. One or two space indentation occurred more frequently.
6. Three or four space indentation occurred more frequently.
7. Five spaces or greater indentation occurred more frequently.
8. Lower case characters only (all source code).
9. Upper case characters only (all source code).
10. Case used to distinguish between keywords and identifiers.
11. Underscore used in identifiers.
12. BEGIN followed by a statement on the same line.
13. THEN followed by a statement on the same line.
14. Multiple statements per line.
15. Blank lines in program body

To test their hypothesis, Cook and Oman collected the metrics mentioned above for eighteen short programs by six authors, each program implementing a simple textbook algorithm that fit on one page.

The programs were taken from example code for three tree traversal algorithms (inorder traversal, preorder traversal and postorder traversal) and one sorting algorithm (bubble sort).

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. This experiment 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.

Their choice of metrics also limits the usefulness of their techniques. Some of these metrics are useless in the analysis of C code because the language is case sensitive and it is a common occurrence that programmers use uppercase for constants and lowercase for variables and identifiers.

A Boolean measurement is inadequate for most of the other measurements. Consider, for example, the metric dealing with the existence of blank lines in a program. Most programmers tend to organize their programs so that some structural information is readily available by simply looking at the program. Programmers then tend to spread their code, using blank lines to separate logically independent blocks. Hence, it is likely that all programs would have at least some blank lines separating their logical components.

To test our theory, we constructed a simple program that counts the blank lines in a program and used this tool on 996 C files. The smallest program (or file) was only two lines long. The largest 6,900 lines long. Table 2.1 contains the statistics gathered. Notice how the distribution of percentages of blank lines is clustered around the ten percent mark. Figure 2.1 shows a graphical representation of this distribution.

Of these, only four programs had no blank lines, and these four files were shorter than six lines of code. The rest varied significantly, from only one blank line to 36% of the file being blank lines. Hence, we conclude that the choice of a Boolean variable for this measurement is inappropriate. Similar arguments can be made for the "Keywords followed by comments," and "Multiple statements per line" measurements.

We also analyzed 178 C files for indentation information. The smallest program (or file) was a few dozen lines. The largest 6,900 lines long. Figure 2.2 shows the distribution of mean indentation values. Figure 2.3 shows the distribution of maximum indentations for a program.

Not surprisingly, most of the programs analyzed have their indentations in the two, four and eight spaces marks. What was surprising was the large

Percent blank lines	Files	Smallest file	Largest file
0	4	2	5
1	4	100	1037
2	3	100	797
3	14	31	1047
4	27	25	1461
5	36	58	6009
6	83	18	3617
7	112	27	2050
8	122	12	3297
9	105	34	2718
10	104	29	2151
11	91	27	2794
12	82	25	5801
13	59	16	1706
14	35	22	1642
15	24	13	719
16	20	19	963
17	12	12	1685
18	22	11	3265
19	6	21	680
20	7	20	92
21	5	29	238
22	3	32	146
23	3	31	212
24	2	38	216
26	4	27	2235
27	1	77	77
28	1	25	25
30	1	23	23
33	2	3	18
36	1	22	22

Table 2.1: Blank Line Count for C Programs

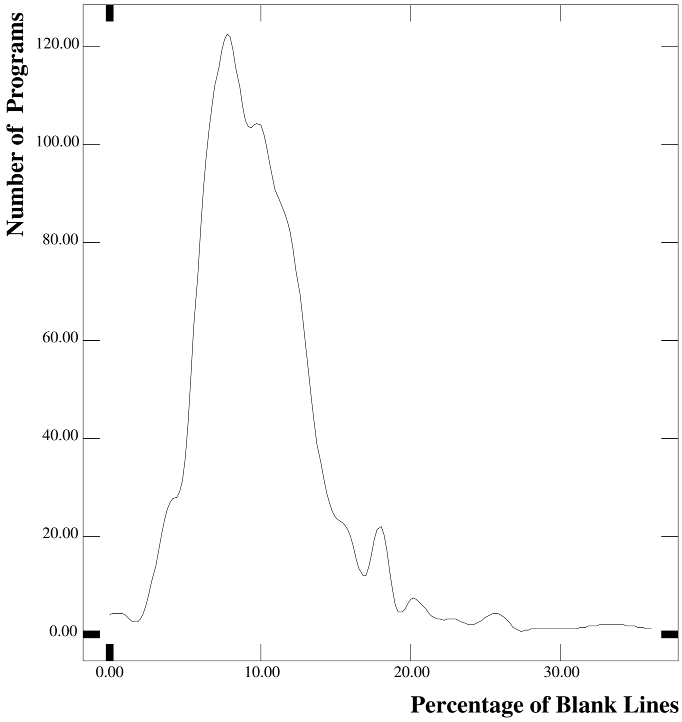

Figure 2.1: Blank Line Distribution in C Programs

variation found. Some programs were overwhelmingly inconsistent, allowing for lines to be indented up to 16 spaces. A better indentation measurement must include a consistency value.

2.2.3 Software Forensics

Spafford and Weeber suggest 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 theorize 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].

The authorship of a program might well be proven beyond any reasonable doubt by the results of such analysis if there exists a large enough statistical

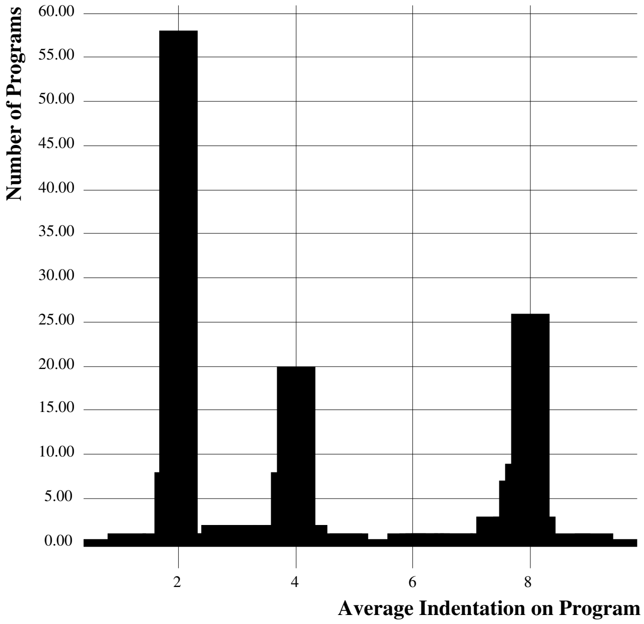

Figure 2.2: Mean Indentation for C Programs

base to support our comparisons. If not, they might provide hints about where more serious investigation should be performed.

An author of software cannot be directly identified by the analysis of source code, much in the same manner as medical forensics cannot directly identify the author of a crime. The result of the analysis will be a series of statistics about the characteristics that are present in the piece of code, much like the autopsy of a murder showing that the victim was stabbed by a 20 inch long knife by a left handed criminal six feet tall.

Spafford and Weeber write in [WS93] the following of software forensics:

> "...would be similar to the use of handwriting analysis by law enforcement officials to identify the authors of documents involved in crimes, or to provide confirmation of the role of a suspect. Handwriting analysis involves identifying features of the writing in question. A feature of the writing can be anything identifiable about the writing, such as the way the i's are dotted or aver-

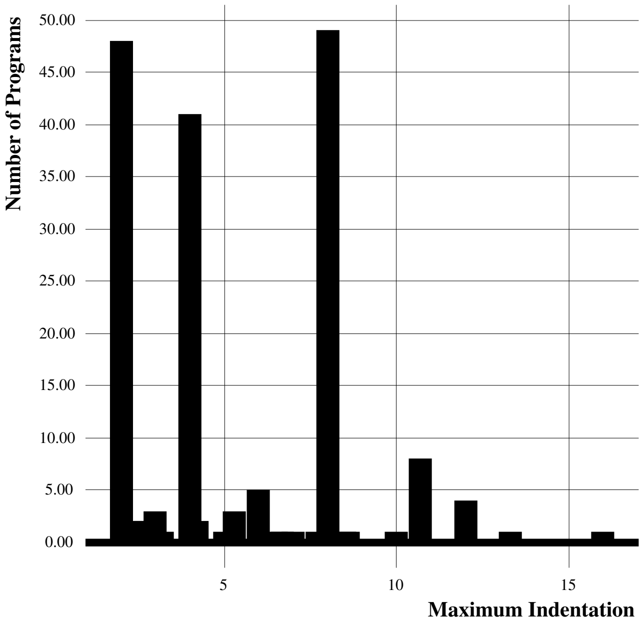

Figure 2.3: Maximum Indentation for C Programs

age height of the characters. The features useful in handwriting analysis are the writer-specific features. A feature is said to be writer-specific if it shows only small variations in the writings of an individual and large variations in the writings of different authors.

Features considered in handwriting analysis today include shape of dots, proportions of lengths, shape of loops, horizontal and vertical expansion of writing, slant, regularity, and fluency. Most features in handwriting are ordinary. However, most writing will also contain features that set it apart from the samples of other authors, features that to some degree are unusual. A sample that contains i's dotted with a single dot probably will not yield much information from that feature. However, if all of the o's in the sample have their centers filled in, that feature may identify the author."

This has a high degree of correlation with the identification of program authors using style analysis [OC89]. However, Software Forensics is really a superset of authorship analysis using style analysis because some of the measurements suggested by Spafford and Weeber [WS93] include, but are not limited to, some of the measurements made by Cook and Oman [OC89].

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 of 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.

The list of measurements suggested by Spafford and Weeber is comprehensive, but the derivation of some of these are difficult to automate. Consider, for example, what they say about spelling and grammar measurements:

> Many programmers have difficulty writing correct prose. Misspelled variable names (e.g. TransactoingReciept) and words inside comments may be quite telling if the misspelling is consistent. Likewise, small grammatical mistakes inside comments or print statements, such as misuse or overuse of em-dashes and semicolons might provide a small, additional point of similarity between two programs.

From a small set of programs, we gathered the names of variable names and found, among others, that the following were not recognized by the local spell checker: talk_w_oracle, om_recv, oracle_mssg, collis_service, result_recv, result_send, net_colls, tcpchksum, pip, sec_host, info_my_socket, msgc, remdata, CharToInputType, StateToNextState, NoOutputChars, ErrorType, proto, addrlen, hname.

We also extracted all the comments and strings from several programs of a graduate student of Purdue University and found that the following words, among others, were not recognized by our local spell checker: algo, buf, cmd, cmdAckGet, cmdAckPut, cmdClr, cmdEnd, cmdErr, cmdGet, cmdPut, co, connectsock, cont, corrida, Entrando, Esto, Expiro, llego, mandando, respuesta, resultado, tiempo, tion, ver.

Even when we look at the original words in their context, it is practically impossible to determine when a word was misspelled. Some of the words the spell checker did not recognize are in Spanish. This gives us some information about the origin or educational background of the programmer [1] and this adds to the complexity of automating these measurements.

Spafford and Weeber provide no statistical evidence that might support their theory. The following quote from the concluding remarks of the paper illustrate the point:

> Further research into this technique, based on the examination of large amounts of code, should provide further insight into the utility of what we have proposed. In particular, studies are needed to 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 should be studied, to determine environment-specific factors that may influence comparisons. And most important, 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.

2.3 Difficulties in Authorship Analysis

We expect the programming characteristics of programmers to change and evolve. For example, it would be unrealistic for us to expect the programming style of a programmer to remain unaltered through several years of graduate work in Computer Science.

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

[1] Some of these words may be in Spanish, or they might be simple contractions. The words ver and algo fall under this category

tools[2]. 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.

2.4 What Authorship Analysis is not

2.4.1 Plagiarism Detection

It is important to realize that authorship analysis is markedly different from Plagiarism Detection. In [Mor91], D. Moreaux defined Software Plagiarism as a general form of software theft, which can in turn be defined as the complete, partial or modified replication of software, with or without the permission of the original author.

Notice that according to this definition, 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.

Consider, for example, a program "X" that is a plagiarized version, by programmer "A" of an original work done by programmer "B." After Programmer "A" has copied the original program, stylistic changes are made. Specifically, old comments are removed and new comments are added, indentation and placement of brackets are changed to match the style of program-

[2]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.

mer “A,” variables are renamed to something that he feels more comfortable with, the order of functions is altered and “for” loops are changed to “while” loops.

While plagiarism detection needs to detect the similarity between these two programs, authorship analysis does not. For the purpose of authorship identification, these two programs have distinct authors.

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. We can, however give a simple example that will illustrate how measurements traditionally used for plagiarism detection are ill suited to authorship analysis.

Consider the two functions shown in figure 2.4. Both are structurally and logically equivalent. However, the second function has undergone several stylistic changes that mask the identity of the original programmer. Plagiarism detection systems might consider them identical. Traditional software engineering metrics, used commonly for plagiarism detection, will yield similar values. But authorship analysis should not consider them identical. If both authors happen to write these functions independently, and this is a common occurrence, our system should identify them as unique.

2.4.2 Personality Profiler

Throughout the development of this document, we realized that it is a common occurrence to mistake authorship analysis with the classification of programmers personalities. A member of the faculty of Purdue University remarked that it would be ridiculous to try to derive any information about his persona by the programming examples in his many published books or developed systems. While this might be possible, it should be left to the researchers in psychology departments.

We also believe that it is possible to determine something about the background of a programmer by looking at his code [WS93]. Faculty members and teaching assistants at Purdue University agree that undergraduate students and electrical engineers are notorious for abusing[3] hashing, Lisp programmers

[3]Abuse in this context refers to using a data structure that is inappropriate or unreasonably expensive. Sorting with linked lists or hashing techniques where keys have a high degree of collisions are examples of such abuses.

are notorious for abusing linked-list data structures, and native Fortran programmers prefer using short lines. While all this information might be useful in forensic analysis [WS93], it is beyond the scope of our study.

2.5 A Simple Example

As a simple example of the differences in programming style among programmers, consider the programs shown in figures 2.5, 2.6, 2.7 and 2.8. These are variations on a program written by graduate students at Purdue University. The specification given for the function was:

> "Write a function that takes three parameters of type pointers to strings. The function must concatenate the first two strings into the third string. The use of system calls is not permitted."

We notice that the approach taken by all programmers was similar; all functions work exclusively with pointers, and all loops search for the null character at the end of each string. Three of the four programs use the while loop and three of the four programs use auxiliary variables. All four programs contain two loops and all loops have the same functionality.

However, these programs are far from being identical. A closer scrutiny of these programs will reveal that:

1. Programmer one prefers the use of "for" loops instead of the "while" loop.

2. Only programmer two has the function header comments boxed in completely, programmer one has partial boxing, and the rest have no boxes at all.

3. Programmers one and three have chosen temporary variables with names of the form xxx1, xxx2, xxx3, etc. Only programer two has chosen temporary variable names that reflect the use that will be given to the variable.

4. One of the programs has a significant bug: The return of the function is undefined when the two input strings are empty (i.e. it fails to "do nothing" gracefully [KP78]).

5. The program for programmer one has an incorrect comment.
6. Only one of the programmers has consistently indented his programs using three spaces. The rest used only two spaces.
7. The placement of curly braces ({}) is distinct for all programmers.

This is not an exhaustive list of differences. It is just an example that illustrates the types of features that we can examine to distinguish among programmers.

```
/********************************************
 * Subroutine for checking a string for TABS
 * Parameters: s1:String to examine
 * Return: Boolean indicating the existence
 *         of a tab character
 ********************************************/
int strchk(char *s1)
{
  char *ptr1;

  ptr1 = s1;
  while(*ptr1 != 0)
    if(*ptr1++ == '\t')
      return(1);
  return(0);
}


#define TRUE 1
#define FALSE 0
/* Checks the existence of \t in string */
int check_for_tab_in_string(string)
char *string;
{
  char *character_pointer;

  /* Loop until found or we reach EOLN */
  for(character_pointer = string;*character_pointer != NULL;)
  {
    /* check to see if we found TAB */
    if(*(character_pointer++) == 9)
    {
      /* Success!! */
      return(TRUE);
    }
  }
  /* No tab */
  return(FALSE);
}
```

Two program modules that might be considered similar in plagiarism detection systems. They would not be considered similar in authorship analysis.

Figure 2.4: Plagiarism Detection vs. Authorship Analysis

```
/***************************************************************
 * This function concatenates the first and second string into
 * the third string.
 ***************************************************************/
void strcat(char *string1, char *string2, char *string3)
{
  char *ptr1, *ptr2;

  ptr2 = string3;
  /*
   * Copy first string
   */
  for(ptr1=string1;*ptr1;ptr1++) {
    *(ptr2++) = *ptr1;
  }
  /*
   * Copy first string
   */
  for(ptr1=string2;*ptr1;ptr12++) {
    *(ptr2++) = *ptr1;
  }
  /*
   * Null terminate the resulting string
   */
  *ptr2 = '\0';
}
```

Figure 2.5: Style Variations on a Program: Programmer 1

```
/* ------------------------------------------------------------------ */
/* strcat(s1, s2, s3)                                                 */
/* Append strings s1 and s2, and copy result into s3.                 */
/* Requires that sufficient memory for s3 is already allocated.       */
/* -------------------------------------------------------------------*/

void strcat (s1, s2, s3)
char *s1;
char *s2;
char *s3;
{
   char* src  = s1;
   char* dest = s3;

   while (*src) {
      *(dest++) = *(src++);
      };

   src = s2;
   while (*src) {
      *(dest++) = *(src++);
      };

   *dest = '\0';
};
```

Figure 2.6: Style Variations on a Program: Programmer 2

```
/*
    str_cat(char *str1, char *str2, char *str3)
    concatenates string str1 & str2 into str3
    */
void str_cat(char *str1, char *str2, char *str3)
{
  char *aux1,* aux2;

  aux1 = str3;
  aux2 = str1;

  while (*aux2 != 0)             /* CIoopy str1 -> str3 */
    {
      *aux1 = *aux2;
      aux1++;
      aux2++;
    }

  aux2 = str2;
  while (*aux2 != 0)             /* Append str23 -> str3 */
    {
      *aux1 = *aux2;
      aux1++;
      aux2++;
    }
                          /* End str3 with a null character */
  *aux1 = 0;
}
```

Figure 2.7: Style Variations on a Program: Programmer 3

```
/*
 * concatenate s2 to s1 into s3.
 * Enough memory for s3 must already be allocated. No checks !!!!!!
 */

mysc(s1, s2, s3)
  char *s1, *s2, *s3;
{
  while (*s1)
    *s3++ = *s1++;

  while (*s2)
    *s3++ = *s2++;
}
```

Figure 2.8: Style Variations on a Program: Programmer 4

Chapter 3

Experimental Setup

We understand that analysis of source code can be performed at many levels. Because programmers solve problems with regular patterns, we could analyze the semantic structure of the code to find repeating patterns and structures. For example, the actions that might be taken on the discovery that some fatal error has occurred might vary considerably from program to program and from programmer to programmer.

Formal methods for defining the semantics, or connotative meaning, of programs and programming languages, such as axiomatic semantics[1] and denotational semantics [All86, Set89, Hoa69], could be used to search for identifying patterns in program logic or semantic structure.

Also, consider a programmer who codes with a top-down approach. Rather than testing each program module individually, his program must be robust enough so that when it is time to test the program, errors will be quickly spotted. In these cases, dynamic error checking might be more frequent than when each module is tested for correctness using the bottom-up approach.

We can also search for repeating patterns by analyzing the executable behavior of the program, searching for data flow patterns, or by looking at user interfaces, looking for repeating patterns in the look and feel of the program.

In our particular environment, it is not likely that we will be able to use these methods. Formal methods like axiomatic semantics are computa-

[1]Axiomatic Semantics concentrates on the predicates that must hold at every step of the program, and deriving meaning from those predicates that hold on termination of the program.

tionally expensive and difficult to automate, and work only for the simplest programs. Analyzing the executable behavior of a program or analyzing user interfaces is possible but, as we shall show in this paper, inappropriate for our environment.

3.1 Choice of Programming Language

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.

Although theoretically possible, it would be impractical to compare style metrics across different development platforms. We must place some restrictions on what must remain constant for the authorship analysis techniques described in this document to work.

The programming language to be used for our analysis should remain the same. 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 LISP, 4th Dimension or Prolog.

Consider, for example, the C program shown in figure 3.2, the Prolog program shown in figure 3.1 and the Emacs-LISP program shown in figure 3.3. 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.

3.2 Definition of Software Metrics

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, are precisely some of these features. Hence, the term software metric,

```
apply( eq , ArgVal , Env , Result ) :-
     ArgVal=[FstArg,ScnArg],
     number( FstArg),number( ScnArg),
     FstArg=ScnArg,Result=t,!.
apply( eq , Arg , Env , Result ) :-
     Arg=[A1,A2],A1=[quote,X],A2=[quote,Y],
     atom(X),atom(Y),X=Y,
     apply(eq,[X,Y],Env,Result),!.
apply( eq , Arg , Env , Result ) :-
     Arg=[A1,A2],
     not( number(A1)),not( number(A2)),
     eval(A1,Env,P1),eval(A2,Env,P2),
     apply( eq , [P1,P2] ,Env ,Result ),!.
apply( eq , ArgVal , Env , Result ) :-
     ArgVal=[FstArg,ScndArg],
     FstArg=[cons,A1,A2],ScndArg=[cons,B1,B2] ,
     apply( eq , [A1,A2], Env, R1),R1=t,
     apply( eq , [B1,B2], Env, Result),!.
apply( eq , ArgVal , Env , Result ) :-
     ArgVal=[FstArg,ScndArg],
     FstArg=[H1|T1],ScndArg=[H2|T2],
     apply( eq , [H1,H2],Env, R1),R1=t,
     append( T1, T2, NewArg),
     apply( eq , NewArg ,Env, Result),!.
```

Figure 3.1: Fragment of a Prolog Program

or simply metric, is more appropriate to describe these special characteristics than the term "marker" used by Cook and Oman in [OC89] or the term "identifying feature" used by Spafford and Weeber [WS93].

3.3 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.

```
main()
{
  char i,j;
  fifo_buffer<char> *buf;

  buf = new fifo_buffer<char>();
  for(i=0;i<MAX_BUFFER_LEN;i++) {
    if(!buf->enter_buffer('A'+i))
      printf("Error: Buffer should not be full!\n");
    if(buf->enter_buffer('A'+10))
      printf("Error: Buffer should be full!\n");
    for(i=0;i<5;i++) {
     if(!buf->leave_buffer(&j))
       printf("Error: Buffer should not be empty!\n");
     else
       printf("( %c ) ",j);
    }
  }
}
```

Figure 3.2: Fragment of a C Program

- D. 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.

3.4 Software Metrics Categories

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

Programming style, as shown by example by Kernighan and Plauger [KP78], Oman and Cook [OC90a] and Ranade and Nash [RN93], is a broad

```
;
; miscellaneous short RCS commands
;
(defun rcs-rlog-file ()
   "Run rlog on this file."
   (interactive)
   (setq this-file (if (dired-mode-p) (dired-get-filename) (buffer-file-name)))
   (message "rlog %s ..." (file-name-nondirectory this-file))
   (sit-for 0)
   (shell-command (concat "rlog " this-file) nil)
   (message "rlog %s ... done" (file-name-nondirectory this-file)))

(defun rcs-diff-file (arg)
   "Run rcsdiff on this file with optional revision"
   (interactive "p")
   (setq this-file (if (dired-mode-p) (dired-get-filename) (buffer-file-name)))
   (setq rcs-diff-rev (rcs-get-revision arg))
   (message "rcsdiff -r%s %s ..." rcs-diff-rev
            (file-name-nondirectory this-file))
   (sit-for 0)
   (shell-command (concat "rcsdiff -r" rcs-diff-rev " " this-file) nil)
   (message "rcsdiff -r%s %s ... done" rcs-diff-rev
            (file-name-nondirectory this-file)))
```

Figure 3.3: Fragment of a Lisp Program

term that encompasses a much greater universe than the one we have chosen in this paper. Programming style generally considers all our three categories.

3.4.1 Programming Layout Metrics

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 alike [MB93].

3.4.2 Programming Style Metrics

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.

3.4.3 Programming Structure 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,

3.5 Metrics Considered

From all the sources mentioned in Section 3.3, 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.

3.5.1 Programming Layout Metrics

- Metric STY1: A vector of metrics indicating indentation style [RN93, pages 68–69]. For example, consider the styles shown in figure 3.4. Our metrics should distinguish among them.

  This vector of metrics will be composed of:

```
if(a==b) {                  if(a==b)
  b = 1;                      {b =1;
}                             }


if(a==b) { b = 1;           if(a==b) { b = 1; )
}


if(a==b) {                  if(a==b)
  b = 1;                    {
  }                           b = 1;
                            }
```

Figure 3.4: Indentation Styles and Placement of Brackets

- 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 ({).

These metrics must only be calculated inside functions, ignoring statements and curly brackets outside them. The formatting of data structure declarations and macro definitions might not represent the programmer's programming style. Also, there is no automatic method for differentiating those declarations and definitions that were created by the user and those that are simply exported from outside modules.

- Metric STY2: Indentation of statements starting with the "else" keyword.

- Metric STY3: In variable declarations, are variable names indented to a fixed column? Figure 3.5 shows an example of variables that are indented to a fixed column.

```
main()
{
  int i;
  char *cptr;
  static float number;
  short int j;
  .
  .
  .
}
```

Un-indented variable declarations

```
main()
{
  int          i;
  char         *cptr;
  static float number;
  short int    j
  .
  .
  .
}
```

Indented variable declarations

Figure 3.5: Variables 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].

3.5.2 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[2] 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.

[2] 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 semantic meaning.

- 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. Here we would specifically search for the "#ifdef" keyword at the beginning of code lines.

- Metric PRO7: Preference of either "while," "for" or "do" loops. All three can be used for the same purposes [KR85].

- Metric PRO8: Does the programmer use comments that are nearly an echo of the code [KP78, page 143] [RN93, page 82]?

- Metric PRO9: Type of function parameter declaration. Does the user use the standard format or the ANSI C format? This metric is only significant if the user has access to an ANSI C compiler.

3.5.3 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[3]. We would specifically be looking at identifiers or constants containing the words "debug" or "dbg" [RN93, pages38–53]. Figure 3.6 shows three common debugging styles in C.

[3]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.

```
#define DBG(x) printf(x)

main() {
  DBG("main\n");
  Parse_Input();
  DBG("Finished\n");
}
```

```
#define DEBUG 1

main() {
#ifdef DEBUG
  printf("main\n");
#endif
  Parse_Input();
#ifdef DEBUG
  DEBUG("Finished\n");
#endif
}
```

```
main(argc, argv)
int argc;
char *argv[];
{
  if(  strcmp(argv[1] == 0 )
    debug = 1;
  else
    debug = 0;
  if(debug) printf("main\n");
  Parse_Input();
  if(debug) printf("Finished\n");
}
```

Figure 3.6: Examples of Debugging Styles

- 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. Alternatively, it could also help identify differences such as

```
root = (-1 * b + sqrt (b*b - 4*a*c)) / (2*a)
```

versus

```
left = -1 * b
right = b*b - 4*a*c
sqright = sqrt (right)
bottom = 2*a
root = (left + sqright) / bottom
```

in which the second programmer has taken 5 lines of code to do what the first programmer did in 1 line of code.

We can define variable count by either looking at each of the statements and counting variables as they are used, or simply by looking at the variable declarations and counting the number of variables declared. The former ignores variables declared and not used and includes global variables. The latter counts all local variables declared and ignores global variables. We have chosen to gather our variable counts by looking at the declarations, ignoring global variables. This allows us to ignore in our calculations those global variables that are imported from external modules like Motif and GKS.

- 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? Surprisingly, 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.

  Consider the case of the function in figure 2.8 where the program has that type of bug. The result of the program is undefined when both input strings are empty.

  However, 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[4] in this metric.

[4]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.

- Metric PSM15: More than any other type of software metric, those that deal with the development phase of 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).

3.6 Computation of Metrics

To find the most accurate metrics we need a series of tools that can collect the subset of metrics mentioned in Section 3.5. Because the tools available calculate only software complexity metrics, we developed a series of programs designed specifically for our purpose and ran them through a series of controlled experiments.

3.6.1 C Source Code Analyzer

Although languages such as Awk can be used for extracting all the necessary metrics [BM85], we avoided the Awk and Perl[WS90] programming languages for calculating most of the metrics dealing with programming structure because existing tools in compiler construction make it easier to write C code, and our programs benefit from the reuse of code that was already tested.

At the heart of the software tools is a software analyzer built for the lcc C compiler front-end developed by David R. Hanson of the University of Princeton [Han91]. An early version of the software analyzer was written by Goran Larsson. This software analyzer collects whatever information it can from the program being analyzed after it passed through the C preprocessor (cpp) and produces an auxiliary list of semantic indentation levels used by other programs; mainly by those programs that calculate metrics regarding indentation.

This list of semantic indentation levels is needed because some C constructs are indented by users following rules different to what the compiler expects. For example, consider the programs in figure 3.7. The semantic indentation generated are the indentation levels according to the compiler. A separate program will later produce from these levels the correct indentation information corresponding to the program as viewed by the user.

3.6.2 Lexical Analyzers

Once the calculation of the programming structure metrics using the modified version of the lcc C compiler have been performed, a series of Perl programs are used to collect the metrics that depend 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.

3.6.3 Statistics Collection

Because the information produced by the above mentioned programs is detailed and numerous, we need a separate series of Perl programs to collect this data, summarize it and generate the statistics mentioned herein. These programs will also generate the formatted data necessary to run statistical analysis and graphic visualization tools.

3.7 Experimental Stages

The experimental data for this paper was gathered in three distinct stages: a preliminary stage, a pilot experiment and a full experiment.

3.7.1 Preliminary Stage

The first stage, or preliminary stage, helped us determine the proper methods for calculating the metrics and coexisted with the tool development phase. For this phase we considered the following subset of the metrics defined in Section 3.5.

We considered the following Programming Layout Metrics defined in Section 3.5.1: mean and standard error for STY1a, STY1b, STY1c, STY1d, STY1e, STY1f, STY1g, mean and standard error for STY1h, mean and standard error for STY1i, mean and standard error for STY2, STY3, STY4, STY5, STY6a, STY6b, STY6c and STY7.

Also considered were the following Programming Style Metrics defined in Section 3.5.2: mean and standard error for PRO1, mean and standard error for PRO2a, mean and standard error for PRO2b, mean and standard error for PRO2c, mean and standard error for PRO2d, PRO3a, PRO3b, PRO3c, PRO3d, PRO4, and PRO5, PRO6, PRO7, PRO8 and PRO9.

Finally, we considered the following Programming Structure Metrics defined in Section 3.5.3: PSM1, PSM2, PSM3, PSM4, PSM5, PSM6, PSM7, PSM8, PSM9 and PSM14. We chose to exclude all metrics dealing with software complexity (metric PSM10) because there is evidence that these are ill suited for our purpose [OC89].

Metrics PSM11a and PSM11b were excluded because we cannot guarantee that all programs tested use such system calls and because it is impossible to detect if the error result of most system calls is being ignored without tracing the execution of the program.

For example, all three program fragments of figure 3.8 check the result of the "malloc" system call for a NULL value (the error code). This can be verified easily for fragment 1. It is more difficult in segment 2 as we have to assume that the routine "fill_buffer" does indeed check for NULL pointers. However, the verification for fragment 3 is complex. This routine, designed to append nodes to an existing linked list, checks for NULL values returned by "malloc" inside the recursive step.

Metric PSM12 was also excluded from our analysis because the notion of "internal representation of data objects" is not well defined and because the metric cannot be extracted without extensive data flow analysis. Consider, for example, the programs shown in figure 3.9. Detecting that the first program relies on the internal representation of the character data type (ASCII in this case) is simple. More complex analysis must be performed to detect that the second example relies on a long integer being 32 bits or longer.

For reasons explained in Section 3.5, we have also chosen to exclude from our analysis metrics PSM13 and PSM15.

The tool that calculates these metrics was applied to a few programs written by graduate students at Purdue University. Other than showing that the metrics were properly calculated, little information can be extracted from the results. The programs were of different lengths, addressed problems in different domains and were sometimes developed over several years. Furthermore, they were not part of a controlled experiment.

3.7.2 Pilot Experiment

To test the metrics chosen in the Preliminary Stage, a pilot experiment was performed with a small number of programmers, each of whom wrote three short and simple programs. 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.

The programmers who volunteered to code these programs were members of a security seminar that met regularly at Purdue University during the summer and fall of 1993. The description of the problem, and the resulting programs were distributed and collected electronically. The descriptions of these programs were:

Program 1 This program must fill a one dimensional array (or vector) of size 1000 with random integer numbers in the range 0-100 and then perform the following actions:

1. Sort the elements in ascending order. The sorting method is not important.
2. Compute the sum of the numbers in the array.
3. Compute the median (or mean) of the numbers in the array.
4. Calculate the frequency of each of the 101 digits (i.e. the number of zeroes in the array, the number of ones in the array, the number of twos in the array, etc.)

5. Print out this information to the screen.

Program 2 The program should store internally the following list:

Age	Weight	Stretch	Muscle	Time
17	140	Neck	Trapezius	5
17	150	Back	Latissimus	6
17	160	Chest	Pectorals	7
17	170	Arm	Tricep	8
17	180	Side	Obliques	9
17	190	Thigh	Quadriceps	10
18	140	Neck	Trapezius	6
18	150	Back	Latissimus	7
18	160	Chest	Pectorals	8
18	170	Arm	Tricep	9
18	180	Side	Obliques	10
18	190	Thigh	Quadriceps	11
19	140	Neck	Trapezius	7
19	150	Back	Latissimus	6
19	160	Chest	Pectorals	7
19	170	Arm	Tricep	8
19	180	Side	Obliques	9
19	190	Thigh	Quadriceps	9
20	140	Neck	Trapezius	4
20	150	Back	Latissimus	6
20	160	Chest	Pectorals	8
20	170	Arm	Tricep	9
20	180	Side	Obliques	10
20	190	Thigh	Quadriceps	9

The program should then proceed to give the user a menu with the following choices:

```
1.- Enter your name
2.- Enter your weight
3.- Enter your age
```

```
4.- Check time
5.- Check muscle
6.- Quit
```

Option 1 enters the name of the user for future reference. Option 2 enters the user's weight (used as index to the table). Option 3 enters the user's age (used as index to the table). Option 4 prompts the user for a body "part" (i.e. Neck, Back, Chest, Arm, Side or Thigh) and prints the time corresponding to the age, weight and body part entered. Option 5 prompts the user for a body "part" (i.e. Neck, Back, Chest, Arm, Side or Thigh) and prints the muscle group corresponding to the body part. All responses from the program should include the name of the user, his age and weight.

Program 3 The program must read from a file a series of integer numbers and print them in reverse order by using an integer stack. The stack must be implemented using a linked list. You must create routines for pushing elements into the stack, popping elements from the stack, creating a stack and deleting a stack. The program must print out the reverse stack right after every element read. For example, the following is a sample run:

```
Read element (1)
Reverse stack is:
                   1
Read element (2)
Reverse stack is:
                   2
                   1
Read element (4)
Reverse stack is:
                   4
                   2
                   1
...
```

3.7.3 Preliminary Analysis

All metrics dealing with global variables were useless in our analysis of the programs as none of the programs examined made use of such variables. Furthermore, subsequent informal polls and the examination of several hundred system utilities and archived programs revealed that it was not likely that many of the programs we would examine in the next stage of our experiment would make heavy use of global variables. Hence, metrics PRO2b, PRO4, PRO5 and PSM7 were ignored in subsequent analysis.

Not surprisingly, metric PSM5, lines of code per function, shows large variations in all our test cases. We do note that this metric appears to be in direct correlation with the problem domain. Program 3 has low values for this metric (two of these are the lowest values for the programmer) and this is to be expected. The routines needed to do the job in this application, create stack, pop, push and delete stack, are naturally short as can be seen in [Coo87, pages 374-375] and [NS86, page 100]. Program 2 has the highest values for this metric, which is logical because this program deals with I/O and user interaction.

Metrics PSM1 and PSM2 also show large variations. However, for one programmer it gives useful information: programmer 3 never uses "void" function definitions. All other programmers do. Hence, metric PSM2 might prove valuable in the future. Metric PSM1 might not be valuable because it might also be correlated to the domain of the application. Consider, for example, program 3. Most of the functions required for the program must use a return type different from "int". Program 2, however, is more likely to have "int" or "void" function definitions.

Surprisingly, metric STY6c, ratio of lines of block style comments to lines of code, shows large variations, but the magnitude of the numbers observed remain constant. Hence it is likely that this metric can be reformulated to consider ranges. (i.e. 0% – 20%, 20% – 30%, etc.) Less variation can be seen in metric STY6b, and this might prove useful in further analysis.

Because the use of goto statements has been virtually banned from the academic community [Dij68, SCS86, RN93, Set89], and because faculty members oppose their use in programming courses, we have chosen to eliminate metric PSM9 from further analysis. The probability of finding these statements in students code at Purdue University is negligible.

Metrics STY1b through STY1g, dealing with placement of curly brackets

({}), are stable. Metric STY1h, however, has proven to be unstable because it is significant only if the curly bracket is consistently the first character in the line. Similar arguments can be made for metric STY1j. Hence, we eliminated these metrics from further analysis.

In this experiment, only a small number of small programs were considered and hence, the statistical base is not large enough to make further observations and little else can be inferred from the results gathered.

```
main()
{
  int i;

  for(i=0;i<3;i++) {
    if( i == 0 )
      statement;
    else if( i == 1 )
      statement;
    else if( i == 2 )
      statement;
    else
      statement;
  }
}
```

The user's perspective of the indentation of the "else if" clause.

```
main()
{
  int i;

  for(i=0;i<3;i++) {
    if( i == 0 )
      statement;
    else
      if( i == 1 )
        statement;
      else
        if( i == 2 )
          statement;
        else
          statement;
  }
}
```

The compiler's perspective of the indentation of the "else if" clause.

Figure 3.7: Differences on Indentation Levels

```
if ( (ptr = malloc(sizeof(struct test))) == NULL ) {
   fprintf(stderr,"Error allocating memory\n");
}
```

1

```
ptr = malloc(MAX_SIZE);
if( fill_buffer(ptr) == SYSERR )
  panic("Can't fill buffer!");
```

2

```
int add_to_list(ptr,num)
struct node *ptr;
int num;
{
  struct node *p;

  if( ptr == NULL )
    panic("Null pointer! Panic");
  if( num <= 1 )
    return;
  p = malloc(sizeof(struct node));
  ptr->next = p;
  add_to_list(p,num-1);
}
```

3

Figure 3.8: Error detection After System Calls - An Example

```
main() {
  char c;

  c = getchar();
  if( (c >= 'A') && (c <= 'Z') )
    c = c + ('a' - 'A');
}
```

```
main() {
  long int i,j;
  for(i=0,j=1;i<32;i++)
    j = j<<1;
}
```

Figure 3.9: Dependency on the Representation of Data Objects

Chapter 4

Full Experiment

4.1 Introduction

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.

4.2 Experiment Setup

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 4.1.

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

We collected several hundred programs by undergraduate students, but almost all were either too small to be useful in our analysis, relied heavily on tools like LEX and YACC, which our software analyzer rejects because they do not conform to ANSI C specifications, were modifications to existing compilers and operating systems, or provided only one program per

Table 4.1: Distribution of Programs for the Complete Experiment

Group Identification	Number of 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

programmer.

Of the programs submitted by the faculty members, half are oriented towards numerical analysis and half oriented towards compiler construction and software engineering.

4.3 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 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.

4.4 Preliminary Analysis and Elimination of Metrics

Not all metrics calculated proved to be useful in further analysis. As mentioned in Section 3.7.2, all metrics dealing with global variables were eliminated because most programs submitted did not use these variables. Likewise, those metrics dealing with "goto" statements and "assert" macro use were eliminated because none of the programs submitted used them.

We want to keep those metrics that show little variation between programs (for a specific programmer) and those metrics that show large variations among programmers. Unfortunately, analysis of the metrics collected show 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 using Matlab version 4.1. The resulting tool presented for each continuous metric (i.e. real valued metric) 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. Specifically, metrics PRO1M, mean for PRO2a, mean for PRO2b, mean for PRO2c, PRO3d, PRO5, PSM1, PSM6, mean for STY1a,

Table 4.2: Classification by Programmer

Programmer	% Classified	Programmer	% Classified
1	100.00	16	100.00
2	100.00	17	100.00
3	33.00	18	0.00
4	100.00	19	71.00
5	100.00	20	100.00
6	77.00	21	100.00
7	77.00	22	33.00
8	100.00	23	100.00
9	100.00	24	75.00
10	100.00	25	20.00
11	77.00	26	0.00
12	100.00	27	100.00
13	100.00	28	100.00
14	100.00	29	25.00
15	50.00		

STY1b, STY1c, STY1d, STY1e, STY1f, mean for STY1i, mean for STY2, STY6b, STY6c, STY7, PRO8, PSM3, STY4 and STY5.

4.5 Experiment Results

4.5.1 Success Rate

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 4.2.

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?" Table 4.3 shows the classification of those programmers whose programs were always identified. Initially we were surprised to see that the programs for seasoned programmers (programmers 10 and 13), a faculty member (programmer 16), 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. As stated in Section 3.7.2, the three programs each programmer developed addressed different problem domains.

3. The programmers who were correctly classified have different backgrounds. This result was unexpected because from our experience grading projects, electrical engineering students are less consistent in their programming style than computer science students and these in turn are less consistent about their programming style than faculty members. One could expect thus to find a greater percentage of matchings among faculty members and computer science students.

4.5.2 Success Rate by Programmer

A small number of programmers were misclassified 30% of the time or less (see Table 4.4. In this category we have the programmer who provided the most programs (seven programs: three for the pilot study, a multi-user chatting program, a lexical analyzer, and two database tools). The rest of the programmers in this category had between three and four programs.

For the programmers who were classified less than 50% of the time (see Figures 4.6 and 4.5,) 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.

As an example of how inconsistent these programmers are, consider the indentation patterns for programmer 18 as shown in Figure 4.1. Not only does the indentation style vary wildly, but sometimes the indentation style has no relationship with the semantic indentation levels as seen in the program fragments 4 and 5.

Table 4.3: Programmers Classified with 100% Accuracy

Programmer	Category	Programmer Class
1	Students 1	Graduate student in Computer Science
2	Pilot 1	Graduate student in Electrical Engineering
4	Students 1	Graduate student in Computer Science
5	Students 1	Graduate student in Computer Science
8	Students 1	Graduate student in Computer Science
9	Students 1	Graduate student in Electrical Engineering
10	Pilot 2	System administrator and security guru
12	Students 1	Graduate student in Electrical Engineering
13	Pilot 2	Graduate student in Computer Science
14	Students 1	Graduate student in Computer Science
16	Faculty	Faculty member of the department of computer science; area of research is numerical analysis
17	Students 1	Graduate student in Electrical Engineering
20	Students 1	Graduate student in Electrical Engineering
21	Students 1	Graduate student in Computer Science
23	Students 1	Graduate student in Mathematics
27	Students 1	Graduate student in Computer Science
28	Students 1	Graduate student in Computer Science

Table 4.4: Programmers Classified with 70% — 100% Accuracy

Programmer	Category	Programmer Class
6	Students 1	Graduate student in Electrical Engineering
7	Students 1	Graduate student in Electrical Engineering
11	Pilot 1	Graduate student in Computer Science
19	Pilot 1 Students 2	Graduate student in Computer Science
24	Students 1	Graduate student in Computer Science

Table 4.5: Programmers Classified with 20% — 50% Accuracy

Programmer	Category	Programmer Class
3	Students 1	Graduate student in Electrical Engineering
15	Students 1	Graduate student in Computer Science
22	Students 1	Graduate student in Computer Science
25	Pilot 1 Students 2	Graduate student in Computer Science
29	Faculty	Faculty member of the department of computer science. Area of research is software engineering and compiler construction

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 posterior inspection of their code revealed. For programmer 26, for example, we could find several characteristics that remained consistent throughout:

1. The programmer used the RCS revision control utilities on all his programs, which of course show his login id.

2. All his comments are unindented one-line comments

3. For every system call that resulted in error, an error message was printed to the standard error using either the fprintf or perror system calls.

4.5.3 Consistency of Classification

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

Table 4.6: Programmers Classified with 0% Accuracy

Programmer	Category	Programmer Class
18	Students 1	Graduate student in Computer Science
26	Students 1	Graduate student in Computer Science

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.

4.5.4 Performance of Metrics

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. We can illustrate this point using a simple example. Consider the metrics shown in Figure 4.2, where the results of two metrics are plotted for three programs for each of four users.

The attentive reader might notice that both metrics show large variations for users three and four and the general usefulness of these metrics individually is limited. However, the combination of these metrics provides unequivocal information to the authorship of the programs tested because both measurements are clustered together, with low or high values, or dispersed.

```
while(condition)
{
if(condition)
{
while(condition)
{
if(condition)
{
statements;
}
}
}
}
```

Fragment 1

```
while(icondition) {
   if(condition) {
      while(condition) {
         if(condition) {
            statements;
         }
      }
   }
}
```

Fragment 2

```
if(condition)
    {
        statements;
    }
else if(nargs == 2)
    {
        statements;
    }
```

Fragment 3

```
statements;
    while(condition)
    {
        statements;
    statements;
    }
statements;
```

Fragment 4

```
for(assignment;condition;increment)
    statement;
    statement;
    statement;
    return(value);
```

Fragment 5

Figure 4.1: Indentation Style Change for Programmer 18

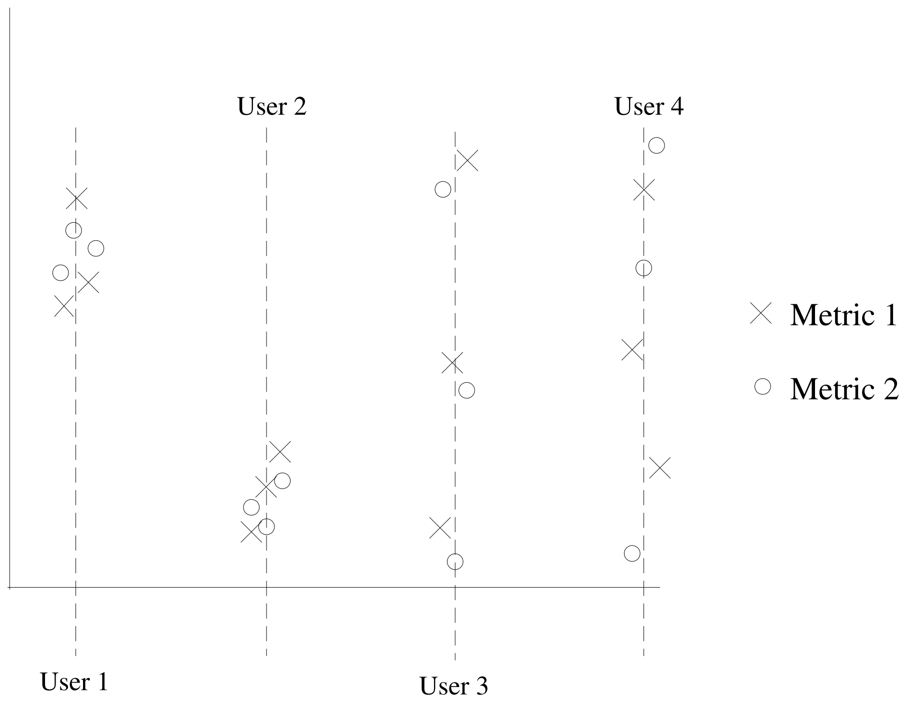

Figure 4.2: Interdependency of Metrics

Chapter 5

Conclusions

5.1 Conclusions

5.1.1 Quality of the Experiment

The first issue that must be resolved is about the quality of the data collected for our experiment. Is the data collected representative of the programming methodologies we are likely to see in real production environments? As stated in section 2.2.2, we noted that the experiment performed by Cook and Oman was fundamentally flawed because the programs analyzed were most likely cleaned and beautified and the algorithms considered addressed the same problem domain.

5.1.2 Existence of Identifying Patterns

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

Table 5.1: Experimental Metrics Subset

	Programmer								
Metric	10	10	10	11	11	11	16	16	16
PRO2b	0	4	0	0	0	0	1	0	0
PRO2c	4	8.3	9	4	4	7.2	8	14	6.5
PRO3d	0	0	0	0	0	0.33	0.5	1	1
PSM1	50	0	0	100	100	66.67	100	100	100
STY1aM	8	7.76	8	1.72	2.41	1.79	4	4	3.97
STY1d	100	100	100	0	0	0	23.81	41.67	93.75
STY1e	100	100	100	100	100	100	23.81	41.67	87.5
STY2M	0	0	0	0	2.5	1	0	0	2.78
STY6b	0	0	0	9.09	16.85	6.32	10.53	10.34	1.83
STY5	2	2	2	1	1	1	1	1	1

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.

Consider, for example the subset of the metrics collected for our experiment that are shown in table 5.1. Even a first glance at these figures will show the repeating patterns that have allowed our statistical analysis to classify most of the programs considered. Programmer 10 has, for example, consistent patterns on metrics STY1aM, STY1d, STY1e, STY6b and STY5. Programmer 11 is also consistent with the same metrics, showing lower values for metrics STY1aM and STY5 and higher values for metric STY6b.

5.1.3 Performance of the Metrics Chosen

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.

By no means do we claim that the set of metrics examined is the only one that might yield stable metrics. 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[1].

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 have not found any experimental evidence of a relation between a program's data structures, length of code, and its efficiency. If software metrics that measure this relationship do exist, they are most likely limited in value for our purposes. As stated before, we are convinced that the analysis of the software complexity metrics, including the analysis of the complexity of data structures, will not yield interesting results.

5.1.4 Evolution of Programming Style

As mentioned in section 2.3, 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 (programmer 16). Further research must be performed to examine the effect that time and experience has on the metrics examined on this document.

[1]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

It would be logical to conclude that for the authorship analysis techniques to work, the metrics would have to be gathered continually over time. As mentioned in Section 1.2, 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.

5.1.5 Classification of Programmers

As mentioned in Section 1.1, we have shown that there exists a mechanism that can be used to ease the recognition of authorship in computer programs. We would like to quote, from Section 1.3, the paragraph that most accurately describes the purpose of the development of this paper:

> 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 ownership of any program.

From the results presented in this paper, it is clear to us that the authorship analysis tools that can be constructed by the examination of source code with the metrics chosen herein might never give a precise identification of the author of the program.

There is a clear parallel between this and the identification of people based on physical descriptions involving weight, height, hair coloring, facial expressions, etc.; certainly many people will match a specific description.

However, much like in courtrooms around the world, a description of a programmer using the metrics presented might be sufficient evidence, or supporting evidence, to prove the identity of a person.

At this stage, we can only classify our metrics as rudimentary and we are now convinced that it is possible for more than one programmer to have the same basic programming style. So, it might not be possible to find a set of metrics that might uniquely identify this programmer.

The results of this paper, however, 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.

5.2 Future Work

In literature, observations are made about the writer's environment to conclude that an author could not have written some literary work. It is undoubtedly true that in computer science, such observations would also be useful. Section 2.2.1 mentions that Mark Twain makes such observations about Shakespeare. Similar observations could be made about a program if we have previous work by a programmer. For example the preference of revision control systems, integrated development environments and the educational background of a programmer might all be used to show that it would be unlikely for the programmer to develop a specific piece of code.

Also, a larger and more comprehensive set of metrics needs to be examined. Many of the metrics that may provide good statistical evidence of the authorship of a program were not calculated in this paper. It would also be interesting to see if a system can be built that would use a different set of metrics for each programmer. Most likely, artificial intelligence or expert systems to search for repeating patterns on the metrics calculated will be needed rather than using discrimination analysis.

During the development of this document, it became apparent that other statistical methods can be used for the analysis of some, or all of the metrics considered herein. In particular, the use of cluster analysis and Bayesian analysis should be investigated, as well as weighting of metrics and the use of prior probabilities.

Finally, if we would like to use programmer identifying characteristics to enhance real-time intrusion detection, more work must be performed in compilers and operating systems. The identifying characteristics must be preserved in binary executables and this information must be protected to prevent alteration.

5.3 Closing Remarks

It was mentioned in Section 1.2 that there are four areas that motivated our research. We recapitulate briefly with supporting evidence as follows:

1. We are convinced that the results presented herein have demonstrated that authorship analysis techniques could be used in courts as support-

ing evidence. However, further research must determine if the metrics considered herein are general enough.

2. In the academic community, unethical copy of programming assignments is a problem that we might partially solve. For one case in our experiment, the authorship analysis tool we developed failed to identify one of the programmers in our data set. We closely examined the programs of this particular programmer and we question that this particular student was the author of all the programs involved.

3. In industry, the techniques presented herein might be used to guarantee that the programmers involved in a project are indeed following a programming methodology.

4. Real-time intrusion detection systems could be enhanced to include authorship information. In environments where rigid programming rules are imposed, the incorporation or compilation of code that was not developed by the user in question might constitute an abnormal use of the system; therefore, security violations could be detected.

   Also, if the authorship information can be incorporated into the executable version of the program, and this information can be protected from tampering by applying digital signatures, the author of executable programs could be traced.

Bibliography

[All86] L. Allison. *A practical introduction to denotational semantics.* Cambridge University Press, first edition, 1986.

[And91] G. R. Andrews. *Concurrent Programming.* The Benjamin/Cummings Publishing Co., first edition, 1991.

[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.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[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.

[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.

[Mor91] D. Moreaux. A formalism for the detection and prevention of illicit program derivations. Master's thesis, Dept. of Computer Science, University of Idaho, 1991.

[MW92] Merriam-Webster. Webster's 7th collegiate dictionary, 1992.

[Nei63] C. Neider. *The Complete Essays of Mark Twain*. Doubleday, 1963.

[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.

[RR83] A. Ralston and E. Reilly. *Encyclopedia of Computer Science and Engineering.* Van Nostrand Reinhold Co., second edition, 1983.

[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.

[Spe83] D. Spencer. *The Illustrated Computer Dictionary.* Merrill Publishing Co., first edition, 1983.

[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.

[WS90] Larry Wall and Randal Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., first edition, 1990.

[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.